ω-Regular Energy Problems

SVEN DZIADEK, Inria, France
ULI FAHRENBERG, EPITA Research Laboratory (LRE), France
PHILIPP SCHLEHUBER-CAISSIER, EPITA Research Laboratory (LRE), France

We show how to efficiently solve problems involving a quantitative measure, here called energy, as well as a qualitative acceptance condition, expressed as a Büchi or Parity objective, in finite weighted automata and in one-clock weighted timed automata. Solving the former problem and extracting the corresponding witness is our main contribution and is handled by a modified version of the Bellman-Ford algorithm interleaved with Couvreur's algorithm. The latter problem is handled via a reduction to the former relying on the corner-point abstraction. All our algorithms are freely available and implemented in a tool based on the open-source platforms TChecker and Spot.

Additional Key Words and Phrases: weighted timed automaton, weighted automaton, energy problem, generalized Büchi acceptance, Parity acceptance, energy constraints

ACM Reference Format:
Sven Dziadek, Uli Fahrenberg, and Philipp Schlehuber-Caissier. 2024. $\boldsymbol{\omega}$-Regular Energy Problems. *Form. Asp. Comput.* 1, 1 (July 2024), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Energy problems in weighted (timed) automata pose the question whether there exist infinite runs in which the accumulated weights always stay positive. Since their introduction in [7], much research has gone into different variants of these problems, for example energy games [12, 18, 30], energy parity games [11], robust energy problems [2], etc., and into their application in embedded systems [19,21], satellite control [5,27], and other areas. Nevertheless, many basic questions remain open and implementations are somewhat lacking.

The above results discuss *looping* automata [31], *i.e.,* ω-automata in which all states are accepting. In practice, looping automata do not suffice because they cannot express all liveness properties. For model checking, formal properties (*e.g.,* in LTL) are commonly translated into (generalized) Büchi automata [9], or Parity automata [29] if determinism is of the issue, which provide models for the class of ω-regular languages.

In this work, we extend energy problems with transition-based generalized Büchi or Parity conditions and treat them for weighted automata as well as weighted timed automata with precisely one clock. On weighted automata we show that they are effectively decidable using a combination of a modified Bellman-Ford algorithm [4, 20] with Couvreur's algorithm [13]. For weighted timed automata we show that one can use the corner-point abstraction [3, 25] to translate the problem to weighted (untimed) automata.

For looping automata, the above problems have been solved in [7]. (This paper also treats energy games and so-called universal energy problems, both of which are of no concern to us here.) While

Authors' addresses: Sven Dziadek, Inria, Paris, France; Uli Fahrenberg, EPITA Research Laboratory (LRE), Paris, France; Philipp Schlehuber-Caissier, EPITA Research Laboratory (LRE), Paris, France.

ACM 0934-5043/2024/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

we can re-use some of the methods of [7] for our Büchi-enriched case, our extension is by no means trivial. First, in the setting of [7] it suffices to find *any* reachable and energy positive loop; now, our algorithm must consider that such loops might not be accepting in themselves but give access to new parts of the automaton which are. Secondly, [7] mostly treats the energy problem with unlimited upper bound, whereas we consider that energy has a ("weak") upper bound beyond which it cannot increase.

In [7] it is claimed that the weak-upper-bound problem can be solved by slight modifications to their solution of the unbounded problem; but this is not the case. For example, the typical Bellman-Ford detection of positive loops might not work when the energy levels attained in the previous step are already equal to the upper bound. Moreover, we argue that the setting considering a weak upper bound is of greater practical interest, as it allows to faithfully model actual physical systems with a bounded capacity to store energy, such as electric vehicles.

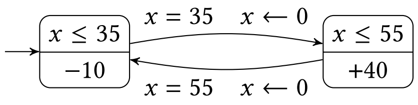
As a second contribution, we have implemented all of our algorithms in a tool based on the open-source platforms TChecker[1] [24] and Spot[2] [14] to solve ω-regular energy problems for one-clock weighted timed automata. We first employ TChecker to compute the zone graph and then use this to construct the corner-point abstraction. This in turn is a weighted (untimed) Büchi or Parity automaton, in which we also may apply a variant of Alur and Dill's Zeno-exclusion technique [1]. Finally, our main algorithm to solve the ω-regular energy problems on weighted finite automata is implemented using a fork of Spot. Our software is available at https://github.com/PhilippSchlehuberCaissier/w

In our approach to solve the latter problem, we do not and cannot fully separate the quantitative constraint on the energy and qualitative acceptance condition (contrary to, for example, [11] which reduces energy parity games to energy games). We first determine the strongly connected components (SCCs) of the unweighted automaton. Then we treat each of the SCCs one by one depending on the acceptance condition. In the case of a generalized Büchi accepting condition, we degeneralize it using the standard counting construction [22]. In the case of the Parity condition, we rely on an approach inspired by a classical algorithm to solve Parity games, devised by Zielonka and published in [32]; the approach presented in [11] uses similar ideas (adapted to their setting). The idea is to work in layers considering the highest priorities first: If the highest priority is accepting, then we treat it much like an accepting transition in the Büchi case. If it is rejecting, we remove the corresponding transitions from the SCC and search in the remaining graph. Finally, we apply a modified Bellman-Ford algorithm to search for energy feasible lassos that start on the main graph and loop on an accepting cycle in the SCC.

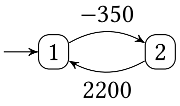
This work is based on our contribution [15] and extends it in two directions. First we generalize the acceptance condition from Büchi to Parity, which allows to represent the class of ω-regular languages with deterministic automata. Secondly we propose an efficient algorithm to compute the actual trace verifying the quantitative and qualitative constraints. This turns out to be a non-trivial task as the structure of these traces is significantly more complicated than those for normal ω-words.

We also correct an error in [15]. There, it is claimed that two iterations suffice to find Büchi accepting and energy feasible cycles. We expose an example where more than two iterations are necessary and where the algorithm given in [15] would fail to find energy feasible cycles. In fact, the number of iterations necessary is linear in the weak upper bound when using that approach. We therefore devise a new algorithm which both corrects this mistake and whose complexity does not depend on the weak upper bound.

[1]See https://github.com/ticktac-project/tchecker
[2]See https://spot.lrde.epita.fr/



(a) Weighted timed automaton A_T



(b) Equivalent (finite) weighted automaton A

Fig. 1. Satellite example: two representations of the base circuit. We mark the acceptance condition of the automaton above its depiction. Here both automata are in fact looping automata, as all infinite runs are accepted.

The rest of the paper is structured as follows. In Section 2 we introduce the energy Büchi problem for finite weighted automata. We also show that these may be degeneralized and that searching for lassos is enough. Section 3 introduces the energy Büchi problem for weighted timed automata and shows how to reduce this to the problem for weighted (untimed) automata. In Section 4 we finish solving the energy Büchi problem for finite weighted automata by developing an algorithm to find feasible lassos. Section 5 shows some benchmarks, and in Section 6 we develop an algorithm to compute the actual trace verifying the energy Büchi constraints in case it exists. Section 7 shows how to reduce energy Parity problems to energy Büchi problems, and Section 8 concludes.

Running example. To clarify notation and put the concepts into context, we introduce a small running example. A satellite in low-earth orbit has a rotation time of about 90 minutes, 40% of which are spent in earth shadow. Measuring time in minutes and (electrical) energy in unspecified "energy units", we may thus model its simplified base electrical system as shown in Figure 1a.

This is a weighted timed automaton (the formalism will be introduced in Section 3) with one clock, x, and two locations. The clock is used to model time, which progresses with a constant rate but can be reset on transitions. The initial location on the left (modeling earth shadow) is only active as long as $x \leq 35$, and given that x is initially zero, this means that the model may stay here for at most 35 minutes. Staying in this location consumes 10 energy units per minute, corresponding to the satellite's base consumption.

After 35 minutes the model transitions to the "sun" location on the right, where it can stay for at most 55 minutes and the solar panels produce 50 energy units per minute, from which the base consumption has to be subtracted. Note that the transitions can only be taken if the clock shows exactly 35 (resp. 55) minutes; the clock is reset to zero after the transition, as denoted by $x \leftarrow 0$. This ensures that the satellite stays exactly 35 minutes in the shadow and 55 minutes in the sun, roughly consistent with the physical reality.

Figure 1b shows a translation of the automaton of Figure 1a to a weighted untimed automaton. State 1 corresponds to the "shadow" location and transitions are annotated with the corresponding weights, the rate of the location multiplied by the time spent in it. In Section 3 we will show how to obtain a weighted automaton from a weighted timed automaton with precisely one clock.

One may now pose the following question: for a given battery capacity b and an initial charge c, is it possible for the satellite to function indefinitely without ever running out of energy? It is clear that for $c < 350$ or $b < 350$, the answer is no: the satellite will run out of battery before ever leaving Earth's shadow; for $b \geq 350$ and $c \geq 350$, it will indeed never run out of energy.

Now assume that the satellite also has some work to do: once in a while it must, for example, send some collected data to earth. Given that we can only handle weighted timed automata with precisely one clock (see Section 3), we model the combined system as in Figure 2. That is, work (modeled by the leftmost location) takes 5 minutes and costs an extra 10 energy units per minute.

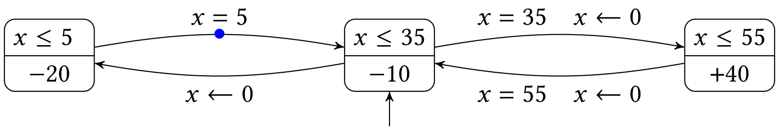


Fig. 2. Weighted timed Büchi automaton A_{T1} for satellite with work module. Only infinite runs containing infinitely many transitions marked by ● are accepted.

The colored dot on the outgoing transition of the work state marks a (transition-based) Büchi condition which forces to take the transition infinitely often in order for the run to be accepted. As a consequence, all accepting runs also visit the "work" state indefinitely often, consistent with the demand to send data once in a while. In order to model the system within the constraints of our modeling formalism, we must make two simplifying assumptions, both unrealistic but conservative:

- work occurs during earth shadow;
- work prolongs earth shadow time.

The reason for the second property is that the clock x is reset to 0 when entering the work state; otherwise we would not be able to model that it lasts 5 minutes without introducing a second clock. It is clear how further work modules may be added in a similar way, each with their own accepting color.

We will come back to this example later and, in particular, argue that the above assumptions are indeed conservative in the sense that any behavior admitted in our model is also present in a more realistic model which we will introduce.

2 ENERGY BÜCHI PROBLEMS IN FINITE WEIGHTED AUTOMATA

We now define energy Büchi problems in finite weighted automata and show how they may be solved. The similar setting for weighted *timed* automata will be introduced in Section 3.

Definition 2.1 (WBA). A *weighted* (transition-based, generalized) *Büchi automaton* (WBA) is a structure $A = (\mathcal{M}, S, s_0, T)$ consisting of a finite set of colors $\mathcal{M}$, a set of states S with initial state $s_0 \in S$, and a set of transitions $T \subseteq S \times 2^{\mathcal{M}} \times \mathbb{R} \times S$.

A transition $t = (s, M, w, s') \in T$ in a WBA is thus annotated by a set of colors M and a real weight w, denoted by $s \xrightarrow{w}_M s'$; to save ink, we may omit any or all of w and M from transitions and $\mathcal{M}$ from WBAs. The automaton A is *finite* if S and $T \subseteq S \times 2^{\mathcal{M}} \times \mathbb{Z} \times S$ are finite (thus finite implies integer-weighted). We use binary encoding for integer weights.

A *run* in a WBA is a finite or infinite sequence $\rho = s_1 \to s_2 \to \cdots$. We write $\mathsf{first}(\rho) = s_1$ for its starting state and, if ρ is finite, $\mathsf{last}(\rho)$ for its final state. *Concatenation* $\rho_1\rho_2$ of runs is the usual partial operation defined if ρ_1 is finite and $\mathsf{last}(\rho_1) = \mathsf{first}(\rho_2)$. Also *iteration* ρ^n of finite runs is defined as usual, for $\mathsf{first}(\rho) = \mathsf{last}(\rho)$, and ρ^ω denotes infinite iteration.

For $c, b \in \mathbb{N}$ [3] and a run $\rho = s_1 \xrightarrow{w_1} s_2 \xrightarrow{w_2} \cdots$, the (c, b)*-accumulated weights* of ρ are the elements of the finite or infinite sequence $\mathsf{weights}_{c\downarrow b}(\rho) = (e_1, e_2, \dots)$ defined by $e_1 = \min(b, c)$ and $e_{i+1} = \min(b, e_i + w_i)$. Hence the transition weights are accumulated, starting with c, but only up to the maximum bound b; increases above b are discarded. We call c the *initial credit* and b the *weak upper bound.*

Running example. In Figure 1b, and choosing $c = 360$ and $b = 750$, we have a single infinite run $\rho = 1 \xrightarrow{-350} 2 \xrightarrow{2200} 1 \xrightarrow{-350} 2 \xrightarrow{2200} 1 \xrightarrow{-350} \cdots$, with $\mathsf{weights}_{c\downarrow b}(\rho) = (360, 10, 750, 400, 750, \dots)$.

[3] Natural numbers include 0.

A run ρ as above is said to be (c,b)*-feasible* if $\mathsf{weights}_{c\downarrow b}(\rho)_i \geq 0$ for all indices i, that is, the accumulated weights of all prefixes are non-negative. (This is the case for the example run above.) For a finite run $\rho = s_1 \xrightarrow{w_1} \cdots \to s_n$ we also write $\mathsf{lastweight}_{c\downarrow b}(\rho) = \mathsf{weights}_{c\downarrow b}(\rho)_n$ for its final accumulated weight, and we will omit the "$\downarrow b$" parts of this notation if no confusion can arise. The following simple fact will prove very useful later.

Lemma 2.2. *For any finite run ρ and $c_1, c_2, b \in \mathbb{N}$ with $c_1 \leq c_2$,* $\mathsf{lastweight}_{c_1}(\rho) \leq \mathsf{lastweight}_{c_2}(\rho) \leq \mathsf{lastweight}_{c_1}(\rho) + c_2 - c_1$.

Proof. For simplicity we may assume $c_2 = c_1 + 1$. If there is an index i such that $\mathsf{weights}_{c_1}(\rho)_i = b$, then $\mathsf{lastweight}_{c_2}(\rho) = \mathsf{lastweight}_{c_1}(\rho)$; otherwise, $\mathsf{lastweight}_{c_2}(\rho) = \mathsf{lastweight}_{c_1}(\rho) + 1$. ☐

An infinite run $\rho = s_1 \to_{M_1} s_2 \to_{M_2} \cdots$ is (generalized, transition-based) *Büchi accepted* if all colors in $\mathcal{M}$ are seen infinitely often along ρ, that is, $\mathcal{M} = \mathrm{Inf}((M_i)_{i\geq 1})$ where

$$\mathrm{Inf}((M_i)_{i\geq 1}) = \{m \in \mathcal{M} \mid \forall i \in \mathbb{N}.\, \exists j \in \mathbb{N}.\, j > i \text{ and } m \in M_j\}.$$

Definition 2.3. The *energy Büchi problem* for a finite WBA A, initial credit $c \in \mathbb{N}$ and weak upper bound $b \in \mathbb{N}$ is to ask whether there exists a Büchi accepted (c,b)-feasible run in A.

Definition 2.4. The *energy Büchi trace problem* for a finite WBA A initial credit $c \in \mathbb{N}$ and weak upper bound $b \in \mathbb{N}$ is to extract a witness run respecting the quantitative and qualitative constraints given that the corresponding energy Büchi problem was answered positively.

These definitions naturally extend to other acceptance conditions like parity. A weighted (transition-based) Parity automaton (WPA) has the same structure as a WBA, however the meaning of the colors changes. Here, each color is assigned to a non-negative integer and we accept all runs for which the largest color seen infinitely often is even. This is commonly called a max-even-Parity condition, more details on this are given in Sec. 7.

Energy problems for finite weighted automata without acceptance condition, asking for the existence of *any* infinite c-feasible run, have been introduced in [7] and extended to multiple weight dimensions in [18] where they are related to vector addition systems and Petri nets. We extend them to (transition-based) generalized Büchi or Parity conditions here but do not consider an extension to multiple weight dimensions.

Degeneralization

As a first step to solving energy problems for finite WBAs, we show that the standard counting construction which transforms generalized Büchi automata into simple Büchi automata with only one color, see for example [22], also applies in our weighted setting. To see that, let $A = (\mathcal{M}, S, s_0, T)$ be a (generalized) WBA, write $\mathcal{M} = \{m_1, \dots, m_k\}$, and define another WBA $\bar{A} = (\bar{\mathcal{M}}, \bar{S}, \bar{s}_0, \bar{T})$ as follows:

$$\bar{\mathcal{M}} = \{m_a\} \qquad \bar{S} = S \times \{1, \dots, k\} \qquad \bar{s}_0 = (s_0, 1)$$
$$\begin{aligned}\bar{T} = &\left\{((s,i), \emptyset, w, (s', i)) \,\middle|\, (s, M, w, s') \in T, m_i \notin M\right\} \\ &\cup \left\{((s,i), \emptyset, w, (s', i+1)) \,\middle|\, i \neq k, (s, M, w, s') \in T, m_i \in M\right\} \\ &\cup \left\{((s,k), \{m_a\}, w, (s', 1)) \,\middle|\, (s, M, w, s') \in T, m_k \in M\right\}\end{aligned}$$

That is, we split the states of A into levels $\{1, \dots, k\}$. At level i, the same transitions exist as in A, except those colored with m_i; seeing such a transition puts us into level $i+1$, or 1 if $i = k$. In the latter case, the transition in $\bar{A}$ is colored by its only color m_a. For succinctness we call such transitions back-edges, since they loop back to the first level of the degeneralization and have a key role in our algorithm. Intuitively, this preserves the language as we are sure that all colors of the original automaton A have been seen:

Lemma 2.5. *For any $c, b \in \mathbb{N}$, A admits a Büchi accepted (c, b)-feasible run iff $\bar{A}$ does.*

Proof. Any infinite run ρ in A translates to an infinite run $\bar{\rho}$ in $\bar{A}$ by iteratively replacing transitions (s, M, w, s') in ρ with the corresponding transitions in $\bar{T}$. Conversely, if $\bar{\rho}$ is an infinite run in $\bar{A}$, then we may replace any transition $((s, i), M, w, (s', j))$ in $\bar{\rho}$ with its preimage in T, yielding a run ρ in A.

Given that the above construction does not affect the weights of transitions, it is clear that ρ is (c, b)-feasible iff $\bar{\rho}$ is. If ρ is Büchi accepted, then m_k is seen infinitely often along ρ, hence m_a is seen infinitely often along $\bar{\rho}$ and $\bar{\rho}$ is Büchi accepted. For the converse, assume that $\bar{\rho}$ is Büchi accepted, then m_a is seen infinitely often along $\bar{\rho}$ and hence m_k is seen infinitely often along ρ.

To finish the proof, we have seen that $\bar{\rho}$ contains infinitely many transitions of the form $((s, k), \{m_a\}, x, (s', 1$
Hence $\bar{\rho}$ also contains infinitely many sequences of transitions

$$(s_1, 1) \longrightarrow \cdots \longrightarrow (s'_1, 1) \longrightarrow (s_2, 2) \longrightarrow \cdots \longrightarrow (s'_2, 2) \longrightarrow \cdots \longrightarrow (s'_{k-1}, k-1) \longrightarrow (s_k, k),$$

traversing all levels of $\bar{A}$. Each of these sequences corresponds, by construction, to a sequence of transitions in ρ along which all of $m_1, \dots, m_{k-1}$ are seen. Hence all of $m_1, \dots, m_{k-1}$ have to be seen infinitely often along ρ and it is thus Büchi accepted. □

Reduction to lassos

An infinite run ρ in A is a *lasso* if $\rho = \gamma_1 \gamma_2^\omega$ for finite runs γ_1 and γ_2. The following lemma shows that it suffices to search for lassos in order to solve energy Büchi problems.

Lemma 2.6. *For any $c, b \in \mathbb{N}$, A admits a Büchi accepted (c, b)-feasible infinite run iff it admits a Büchi accepted (c, b)-feasible lasso.*

Proof. By degeneralization we may assume that A has only one color. Let ρ be a Büchi accepted (c, b)-feasible run in A. Assume first that there exist runs γ_1, γ_2 and ρ' (the first two finite and the last infinite) such that $\rho = \gamma_1 \gamma_2 \rho'$, $\mathsf{first}(\gamma_2) = \mathsf{last}(\gamma_2)$ (*i.e.*, γ_2 is a cycle), γ_2 visits an accepting transition, and $\mathsf{lastweight}_0(\gamma_2) \geq 0$. Using the first inequality of Lemma 2.2, $\mathsf{lastweight}_{\bar{c}}(\gamma_2) \geq 0$ for any $\bar{c} \in \mathbb{N}$, so $\bar{\rho} = \gamma_1 \gamma_2^\omega$ is a Büchi accepted feasible lasso.

Now assume that there is no cycle γ_2 as above. Let $t \in T$ be an accepting transition such that ρ visits t infinitely often and write $\rho = \gamma' t \gamma_1 t \gamma_2 \dots$ as an infinite concatenation of finite runs. Then all $t\gamma_i$ are cycles which visit an accepting transition. By our assumption they must thus all satisfy $\mathsf{lastweight}_0(t\gamma_i) < 0$, *i.e.*, $\mathsf{lastweight}_0(t\gamma_i) \leq -1$. Using the second inequality of Lemma 2.2, $\mathsf{lastweight}_{\bar{c}}(t\gamma_i) \leq \bar{c} - 1$ for all $\bar{c} \in \mathbb{N}$. Let $c' = \mathsf{lastweight}_c(\gamma')$, then $\mathsf{lastweight}_c(\gamma' t \gamma_1 \dots t \gamma_{c'+1}) < 0$ in contradiction to c-feasibility of ρ. □

Hence our energy Büchi problem may be solved by searching for Büchi accepted c-feasible lassos. We detail how to do this in Section 4, here we just sum up the complexity result which we prove at the end of Section 4.

Theorem 2.7. *Energy Büchi problems for finite WBA are decidable in polynomial time.*

3 ENERGY BÜCHI PROBLEMS FOR WEIGHTED TIMED AUTOMATA

We now extend our setting to weighted timed automata. Let X be a finite set of clocks. We denote by $\Phi(X)$ the set of *clock constraints* φ on X, defined by the following grammar:

$$\varphi ::= x \bowtie k \mid \varphi_1 \land \varphi_2 \qquad \big(x \in X, k \in \mathbb{N}, \bowtie \in \{\leq, <, \geq, >, =\}\big)$$

A *clock valuation* on X is a function $v : X \to \mathbb{R}_{\geq 0}$. The clock valuation v_0 is given by $v_0(x) = 0$ for all $x \in X$, and for $v : X \to \mathbb{R}_{\geq 0}$, $d \in \mathbb{R}_{\geq 0}$, and $R : X \to (\mathbb{N} \cup \{\perp\})$, we define the delay $v + d$ and reset $v[R]$ by

$$(v+d)(x) = v(x) + d, \qquad v[R](x) = \begin{cases} v(x) & \text{if } R(x) = \perp, \\ R(x) & \text{otherwise.} \end{cases}$$

Note that in $v[R]$ we allow clocks to be reset to arbitrary non-negative integers instead of only 0 which is assumed in most of the literature. It is known [26] that this does not change expressivity, but it adds notational convenience. A clock valuation v *satisfies* clock constraint φ, denoted $v \models \varphi$, if φ evaluates to true with x replaced by $v(x)$ for all $x \in X$.

Definition 3.1 (WTBA). A *weighted timed* (transition-based, generalized) *Büchi automaton* (WTBA) is a structure $A = (\mathcal{M}, Q, q_0, X, I, E, r)$ consisting of a finite set of colors $\mathcal{M}$, a finite set of locations Q with initial location $q_0 \in Q$, a finite set of clocks X, location invariants $I : Q \to \Phi(X)$, a finite set of edges $E \subseteq Q \times 2^{\mathcal{M}} \times \Phi(X) \times (\mathbb{N} \cup \{\perp\})^X \times Q$, and location weight-rates $r : Q \to \mathbb{Z}$.

As before, we may omit $\mathcal{M}$ from the signature and colors from edges if they are not necessary in the context. Note that the edges carry no weights here, which would correspond to discrete weight updates. In a WTBA, only locations are weighted by a rate. Even without Büchi conditions, the approach laid out here would not work for weighted edges. This was already noted in [7]; instead it requires different methods which are developed in [6] (see also [16,17]). There, one-clock weighted timed automata (with edge weights) are translated to finite automata weighted with so-called *energy functions* instead of integers. We believe that our extension to Büchi conditions should also work in this extended setting, but leave the details to future work.

The *semantics* of a WTBA A as above is the (infinite) WBA $[\![A]\!] = (\mathcal{M}, S, s_0, T)$ given by $S = \{(q, v) \in Q \times \mathbb{R}^X_{\geq 0} \mid v \models I(q)\}$ and $s_0 = (q_0, v_0)$. Transitions in T are of the following two types:

- *delays* $(q, v) \xrightarrow{w}{}^{d}_{\emptyset} (q, v + d)$ for all $(q, v) \in S$ and $d \in \mathbb{R}_{\geq 0}$ for which $v + d' \models I(q)$ for all $d' \in [0, d]$, with $w = r(q)d$; [4]
- *switches* $(q, v) \xrightarrow{0}{}^{0}_{M} (q', v')$ for all $e = (q, M, g, R, q') \in E$ for which $v \models g$, $v' = v[R]$ and $v' \models I(q')$.

Each state in $[\![A]\!]$ corresponds to a tuple containing a location in A and a clock valuation $X \to \mathbb{R}_{\geq 0}$. This allows to keep track of the discrete state as well as the evolution of the clocks. By abuse of notation, we will sometimes write $(q, v) \in [\![A]\!]$ instead of $(q, v) \in S$, for S as defined above.

We may now pose energy Büchi problems also for WTBAs, but we wish to exclude infinite runs in which time is bounded, so-called Zeno runs. Formally an infinite run $(q_0, v_0) \to^{d_1} (q_1, v_1) \to^{d_2} \cdots$ is *Zeno* if $\sum d_i$ is finite: Zeno runs admit infinitely many steps in finite time and are hence considered unrealistic from a modeling point of view [1,23].

Definition 3.2. The *energy Büchi problem* for a WTBA A, initial credit $c \in \mathbb{N}$ and weak upper bound $b \in \mathbb{N}$ is to ask if there exists a Büchi accepted (c, b)-feasible non-Zeno run in $[\![A]\!]$.

We continue our running example; but to do so properly, we need to introduce products of WTBAs. Let $A_i = (\mathcal{M}_i, Q_i, q_0^i, X_i, I_i, E_i, r_i)$, for $i \in \{1, 2\}$, be WTBAs. Their *product* is the WTBA

[4]Here we annotate transitions with the time d which passes; we only need this to exclude Zeno runs below and will otherwise omit the annotation.

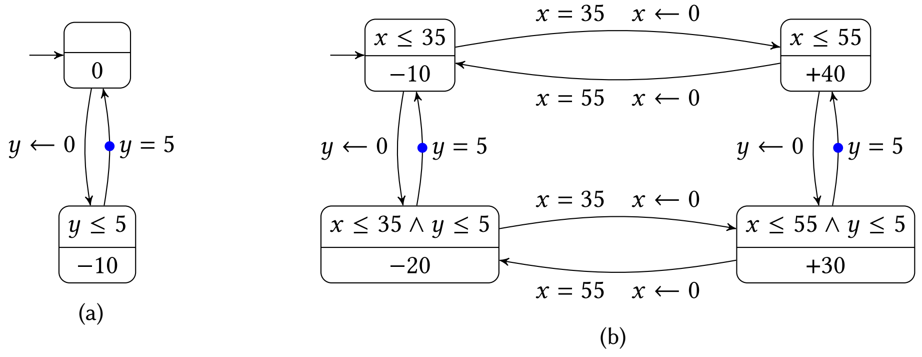

0
$y \leftarrow 0$
$y = 5$
$y \leq 5$
-10
(a)
$x \leq 35$
-10
$x = 35 \quad x \leftarrow 0$
$x \leq 55$
$+40$
$x = 55 \quad x \leftarrow 0$
$y \leftarrow 0$
$y = 5$
$y \leftarrow 0$
$y = 5$
$x = 35 \quad x \leftarrow 0$
$x \leq 35 \wedge y \leq 5$
-20
$x \leq 55 \wedge y \leq 5$
$+30$
$x = 55 \quad x \leftarrow 0$
(b)

Fig. 3. Satellite example. (a) work module W; (b) product $B_1 = A \parallel W$

$A_1 \parallel A_2 = (\mathcal{M}, Q, q_0, X, I, E, r)$ with

$$\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2, \qquad Q = Q_1 \times Q_2, \qquad q_0 = (q_0^1, q_0^2), \qquad X = X_1 \cup X_2,$$
$$I((q_1, q_2)) = I(q_1) \wedge I(q_2), \qquad r((q_1, q_2)) = r(q_1) + r(q_2),$$
$$E = \big\{((q_1, q_2), M, g, R, (q_1', q_2)) \bigm| (q_1, M, g, R, q_1') \in E_1\big\}$$
$$\cup \big\{((q_1, q_2), M, g, R, (q_1, q_2')) \bigm| (q_2, M, g, R, q_2') \in E_2\big\}.$$

Running example. Let A be the basic WTBA of Figure 1a and A_1 the combination of A with the work module of Figure 2. Now, instead of building A_1 as we have done, a principled way of constructing a model for the satellite-with-work-module would be to first model the work module W and then form the product $A \parallel W$. We show such a work module and the resulting product B_1 in Figure 3.

As expected, W expresses that work takes 5 minutes and costs 10 energy units per minute, and the Büchi condition enforces that work is executed infinitely often. The product B_1 models the shadow-sun phases together with the fact that work may be executed at any time, and contrary to our "unrealistic" model A_1 of Figure 2, work does not prolong earth shadow time.

Now B_1 has *two* clocks, and we will see below that our constructions can handle only one. This is the reason for our "unrealistic" model A_1, and we can now state precisely in which sense it is conservative: if $[\![A_1]\!]$ admits a Büchi accepted c-feasible non-Zeno run, then so does $[\![B_1]\!]$. For a proof of this fact, one notes that any infinite run ρ in $[\![A_1]\!]$ may be translated to an infinite run $\bar\rho$ in $[\![B_1]\!]$ by adjusting the clock valuation by 5 whenever the work module is visited.

Bounding Clocks

As a first step to solve energy Büchi problems for WTBAs, we show that we may assume that the clocks in any WTBA A are bounded above by some $N \in \mathbb{N}$, *i.e.*, such that $v(x) \leq N$ for all $(q, v) \in [\![A]\!]$ and $x \in X$. This is shown for reachability in [3]; the following lemma extends it to Büchi acceptance.

Lemma 3.3. *Let $A = (\mathcal{M}, Q, q_0, X, I, E, r)$ be a WTBA and $c, b \in \mathbb{N}$. Let N the maximum constant appearing in any invariant $I(q)$, for $q \in Q$, or in any guard g, for $(q, M, g, R, q') \in E$. There is a WTBA $\bar A = (\mathcal{M}, Q, q_0, X, \bar I, \bar E, r)$ such that*

1. *$v(x) \leq N + 2$ for all $x \in X$ and $(q, v) \in [\![\bar A]\!]$, and*
2. *there exists a (c, b)-feasible Büchi accepted run in $[\![A]\!]$ iff such a run exists in $[\![\bar A]\!]$.*

Proof. Following [3] we define

$$\bar{E} = E \cup \big\{(q, \emptyset, (x = N+2), (x \leftarrow N+1), q) \bigm| q \in Q, x \in X\big\},$$
$$\bar{I}(q) = I(q) \wedge \textstyle\bigvee_{x \in X}(x \le N+2),$$

that is, clock values are reset to $N+1$ whenever they reach $N+2$ using uncolored transitions. Hence $\bar{A}$ satisfies the first requirement.

Let $\cong \subseteq \mathbb{R}_{\ge 0}^X \times \mathbb{R}_{\ge 0}^X$ be the relation on clock valuations defined by

$$v \cong v' \quad \text{iff} \quad \forall x \in X : v(x) \le N \implies v(x) = v'(x), v(x) > N \iff v'(x) > N.$$

Using the same proof as in [3], we may show that for any states (q, v), (q', v') in $[\![A]\!]$ and any finite run $\rho : (q, v) \xrightarrow{w_1} \cdots \xrightarrow{w_n} (q', v')$, there exists a finite run $\bar{\rho} : (q, v) \xrightarrow{w_1'} \cdots \xrightarrow{w_m'} (q', v'')$ in $[\![\bar{A}]\!]$ with $v' \cong v''$ and $\mathsf{lastweight}_c(\rho) = \mathsf{lastweight}_c(\bar{\rho})$, and vice versa: $\bar{\rho}$ is constructed from ρ by inserting special reset transitions when appropriate, and ρ from $\bar{\rho}$ by removing them.

Using the above procedure, any infinite run ρ in $[\![A]\!]$ may be iteratively converted to an infinite run $\bar{\rho}$ in $[\![\bar{A}]\!]$ and vice versa. It is clear that ρ is Büchi accepted iff $\bar{\rho}$ is, and that ρ is (c, b)-feasible iff $\bar{\rho}$ is. □

Corner-point abstraction

We now restrict to WTBAs with only *one* clock and show how to translate these into finite untimed WBAs using the corner-point abstraction. This abstraction may be defined for any number of clocks, but it is shown in [8] that the energy problem is undecidable for weighted timed automata with four clocks or more; for two or three clocks the problem is open.

Let $A = (\mathcal{M}, Q, q_0, X, I, E, r)$ be a WTBA with $X = \{x\}$ a singleton. Using Lemma 3.3 we may assume that x is bounded by some $N \in \mathbb{N}$, *i.e.*, such that $v(x) \le N$ for all $(q, v) \in [\![A]\!]$.

Let $\mathfrak{C}$ be the set of all constants which occur in invariants $I(q)$ or guards g or resets R of edges (q, M, g, R, q') in A, and write $\mathfrak{C} \cup \{N\} = \{a_1, \dots, a_{n+1}\}$ with ordering $0 \le a_1 < \cdots < a_{n+1}$. The *corner-point regions* [3, 25] of A are the subsets $\{a_i\}$, for $i = 1, \dots, n+1$, $[a_i, a_{i+1}[$, and $]a_i, a_{i+1}]$, for $i = 1, \dots, n$, of $\mathbb{R}_{\ge 0}$; that is, points, left-open, and right-open intervals on $\{a_1, \dots, a_{n+1}\}$.

These are equivalent to clock constraints $x = a_i$, $a_i \le x < a_{i+1}$, and $a_i < x \le a_{i+1}$, respectively, defining a notion of implication $\mathfrak{r} \implies \varphi$ for $\mathfrak{r}$ a corner-point region and $\varphi \in \Phi(\{x\})$.

The corner-point abstraction of A is the finite WBA $\mathsf{cpa}(A) = (\mathcal{M} \cup \{m_z\}, S, s_0, T)$, where $m_z \notin \mathcal{M}$ is a new color, $S = \{(q, \mathfrak{r}) \mid q \in Q, \mathfrak{r} \text{ corner-point region of } A, \mathfrak{r} \implies I(q)\}$, $s_0 = (q_0, \{0\})$, and transitions in T are of the following types:

- *delays* $(q, \{a_i\}) \xrightarrow{0}_{\emptyset} (q, [a_i, a_{i+1}[)$, $(q, [a_i, a_{i+1}[) \xrightarrow{w}_{\{m_z\}} (q,]a_i, a_{i+1}])$ with $w = r(q)(a_{i+1} - a_i)$, and $(q,]a_i, a_{i+1}]) \xrightarrow{0}_{\emptyset} (q, a_{i+1})$;
- *switches* $(q, \mathfrak{r}) \xrightarrow{0}_M (q', \mathfrak{r})$ for $e = (q, M, g, (x \mapsto \bot), q') \in E$ with $\mathfrak{r} \implies g$ and $(q, \mathfrak{r}) \xrightarrow{0}_M (q', \{k\})$ for $e = (q, M, g, (x \mapsto k), q') \in E$ with $\mathfrak{r} \implies g$.

The new color m_z is used to rule out Zeno runs, see [1] for a similar construction: any Büchi accepted infinite run in $\mathsf{cpa}(A)$ must have infinitely many time-increasing delay transitions $(q, [a_i, a_{i+1}[) \xrightarrow{w}_{\{m_z\}} (q,]a_i, a_{i+1}])$.

Theorem 3.4. *Let A be a one-clock WTBA and $c \in \mathbb{N}$.*

(1) *If there is a non-Zeno Büchi accepted c-feasible run in $[\![A]\!]$, then there is a Büchi accepted c-feasible run in* $\mathsf{cpa}(A)$.
(2) *If there is a Büchi accepted c-feasible run in* $\mathsf{cpa}(A)$, *then there is a non-Zeno Büchi accepted $(c + \varepsilon)$-feasible run in $[\![A]\!]$ for any $\varepsilon > 0$.*

The so-called *infimum energy condition* [7] in the second part above, replacing c with $c+\varepsilon$, is necessary in the presence of *strict* constraints $x<c$ or $x>c$ in A. The proof maps runs in A to runs in $\mathrm{cpa}(A)$ by pushing delays to endpoints of corner-point regions, ignoring strictness of constraints, and this has to be repaired by introducing the infimum condition.

Proof. To show the first part, let ρ be a non-Zeno Büchi accepted c-feasible run in $[\![A]\!]$. We follow [7] and convert ρ to an infinite run $\bar\rho$ in $\mathrm{cpa}(A)$ by pushing delay transitions within a common corner-point region to the most profitable endpoints. Let $k\in\{1,\dots,n\}$ and consider a maximal subsequence

$$(q_1,v_1)\xrightarrow{w_1}\cdots\xrightarrow{w_{m-1}}(q_m,v_m)$$

of ρ for which $v_1,\dots,v_m\in\,]a_k,a_{k+1}[$. Then all switch transitions in this sequence are non-resetting.

Let $(q'_1,\dots,q'_p)$ be the subsequence of $(q_1,\dots,q_m)$ of first-unique elements, that is, $(q'_1,\dots,q'_p)=(q_{i_1},\dots,q_{i_p})$ is such that $q_{i_1}=q_1=\cdots=q_{i_2-1}\neq q_{i_2}=\cdots=q_{i_3-1}$ etc. Let $j\in\{1,\dots,p\}$ be such that $r(q'_j)$ is maximal, then we construct the following finite run in $\mathrm{cpa}(A)$:

$$(q'_1,[a_k,a_{k+1}[)\xrightarrow{0}\cdots\xrightarrow{0}(q'_j,[a_k,a_{k+1}[)\xrightarrow{w}(q'_j,]a_k,a_{k+1}])\xrightarrow{0}\cdots\xrightarrow{0}(q'_p,]a_k,a_{k+1}]),$$

with $w=r(q'_j)\,(a_{k+1}-a_k)$. (This is possible as there are no guards or invariants in the interval $]a_k,a_{k+1}[$.)

We have seen how to convert maximal finite non-resetting sub-runs of ρ to runs in $\mathrm{cpa}(A)$, so all we have left to treat are resetting switch transitions $(q,v)\xrightarrow{0}(q',x\leftarrow k)$; these are converted to transitions $(q,\mathfrak{r})\xrightarrow{0}(q',\{k\})$. This finishes the construction of $\bar\rho$.

Now $\bar\rho$ sees all the colors in $\mathcal{M}$ infinitely often because ρ does. Given that we have maximized energy gains when converting ρ to $\bar\rho$ (by pushing delays to the most profitable location), $\bar\rho$ is also c-feasible. Finally, ρ being non-Zeno implies that also the color m_z is seen infinitely often in $\bar\rho$.

For the other direction, let K be the maximum absolute value of the weight-rates of A and $\bar\rho=(q_0,\{0\})t'_1t'_2\cdots$ a Büchi accepted c-feasible run in $\mathrm{cpa}(A)$. We iteratively construct an infinite run $\rho=(q_0,0)t_1t_2\cdots$ in $[\![A]\!]$; we have to be careful with region boundaries because of potential strict constraints in A.

Assume that $t_1\cdots t_{n-1}(q,v)$ has been constructed.

- If t'_n is a switch $(q,\mathfrak{r})\xrightarrow{0}_M(q',\mathfrak{r})$, we let $t_n=(q,v)\xrightarrow{0}_M(q',v)$.
- If t'_n is a switch $(q,\mathfrak{r})\xrightarrow{0}_M(q',\{k\})$, we let $t_n=(q,v)\xrightarrow{0}_M(q',v[x\mapsto k])$.
- If t'_n is a delay $(q,\{a_i\})\xrightarrow{0}_\emptyset(q,[a_i,a_{i+1}[)$, then $v(x)=a_i$ and we let $t_n=(q,v)\xrightarrow{w}_\emptyset(q,v+d)$ with $d=\frac{\varepsilon}{2^{n+1}K}$ and $w=r(q)d$. (We must introduce a delay $\frac{\varepsilon}{2^{n+1}K}$ here given that the constraint in q may be strict; but we keep it sufficiently small so that in the end, the sum of all such new delays is bounded above.)
- If t'_n is a delay $(q,[a_i,a_{i+1}[)\xrightarrow{w'}_{\{m_z\}}(q,]a_i,a_{i+1}])$, then by construction, $v(x)=a_i+\frac{\varepsilon}{2^mK}$ for some $1\le m\le n$, and we let $t_n=(q,v)\xrightarrow{w}_\emptyset(q,v+d)$ with $d=a_{i+1}-a_i-\frac{\varepsilon}{2^{n+1}K}-\frac{\varepsilon}{2^mK}$ and $w=r(q)d$.
- If t'_n is a delay $(q,]a_i,a_{i+1}])\xrightarrow{0}_\emptyset(q,a_{i+1})$, then by construction, $v(x)=a_{i+1}-\frac{\varepsilon}{2^mK}$ for some $1\le m\le n$, and we let $t_n=(q,v)\xrightarrow{w}_\emptyset(q,v+d)$ with $d=\frac{\varepsilon}{2^mK}$ and $w=r(q)d$.

If is clear that ρ is Büchi accepted. Given that $\bar\rho$ is $c+\varepsilon$-feasible and the differences in delays between $\bar\rho$ and ρ are bounded by $\sum_{n=1}^{\infty}\frac{\varepsilon}{2^nK}=\frac{\varepsilon}{K}$, it is clear that ρ is $c+\varepsilon$-feasible. □

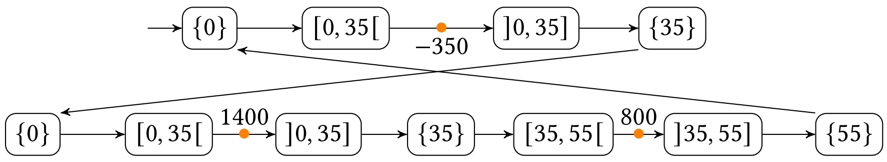


Fig. 4. Corner-point abstraction of base module of Figure 1a.

Running example. We construct the corner-point abstraction of the base module A of Figure 1a. Its constants are $\{0, 35, 55\}$, yielding the following corner point regions:

$$\{0\}, \quad [0, 35[, \quad]0, 35], \quad \{35\}, \quad [35, 55[, \quad]35, 55], \quad \{55\}$$

The corner-point abstraction of A now looks as in Figure 4, with the states corresponding to the "shadow" location in the top row; the colored transitions correspond to the ones in which time elapses. Note that this WBA is equivalent to the one in Figure 1b.

Using the corner-point abstraction, we may now solve energy Büchi problems for one-clock WTBAs by translating them into finite WBAs and applying the algorithms of Section 2 and the forthcoming Section 4. Note that as we only have one clock, the size of the corner-point abstraction is linear in the size of the input WTBA.

4 IMPLEMENTATION

We now describe our algorithm to solve energy Büchi problems for finite WBA before detailing the changes necessary to treat Parity automata. All of this has been implemented and is available at https://github.com/PhilippSchlehuberCaissier/wspot.

We have seen in Section 2 that this problem is equivalent to the search for Büchi accepted (c, b)-feasible lassos. By definition, a lasso $\rho = \gamma_1\gamma_2^{\omega}$ consists of two parts, the lasso prefix γ_1 (possibly empty, only traversed once) and the lasso cycle γ_2 (repeated indefinitely). In order for ρ to be Büchi accepted and (c, b)-feasible the following constraints need to hold:

- the prefix γ_1 must be (c, b)-feasible;
- the cycle γ_2 must be $(\mathsf{lastweight}_c(\gamma_1), b)$-feasible;
- the cycle γ_2 must be $(\mathsf{lastweight}_c(\gamma_1\gamma_2^i), b)$-feasible for all $i > 0$.

The first constraint ensures that the prefix is energy feasible. The second constraint ensures that we can take the cycle once after traversing the prefix. The third constraint expresses the need to loop in γ_2 indefinitely. In fact, the energy $\mathsf{lastweight}_c(\gamma_1)$ may be greater than $\mathsf{lastweight}_c(\gamma_1\gamma_2)$, however after sufficiently many traversals of γ_2 the energy must stabilize (that is $\mathsf{lastweight}_c(\gamma_1\gamma_2^m) = \mathsf{lastweight}_c(\gamma_1\gamma_2^{m+1})$ for some sufficiently large m), while remaining (c, b)-feasible at all times. As it turns out it is quite tricky to get this part correct and we will discuss it in greater detail later on.

Finally the cycle γ_2 obviously needs to be Büchi accepted.

Finding lassos

The overall procedure to find lassos is described in Algorithm 1. It is based on two steps. In step one we compute all energy-optimal paths starting at the initial state of the automaton with initial credit c. This step is done on the original WBA, and we do not take into account the colors. Optimal paths found in this step will serve as lasso prefixes.

The second step is done individually for each Büchi accepting SCC. The COUVREUR algorithm, used to identify the SCCs, ignores the weights, and we can use the version distributed by Spot. We then degeneralize the accepting SCCs one by one, as described in Section 2; recall that this creates

Algorithm 1 Algorithm to find Büchi accepted lassos in WBA

```
Input: weak upper bound $b$
1: function BüchiEnergy(graph $G$, initial credit $c$)
2:     $E \leftarrow$ FindMaxE($G, G.initial\_state, c$)        // $E\colon S \to \mathbb{N}$, mapping states to energy
3:     $SCCs \leftarrow$ Couvreur($G$)                            // Find all SCCs
4:     for all $scc \in SCCs$ do
5:         $GS, back\text{-}edges \leftarrow degeneralize(scc)$
6:         for all $be = src \xrightarrow{w} dst \in back\text{-}edges$ do
7:             $E' \leftarrow$ FindMaxE($GS, dst, E[dst]$)          // $be.dst$ is in $G$ and $GS$...
8:             $e' \leftarrow \min(b, E'[src] + w)$                  // ...(see Figure 5b)
9:             if $E[dst] \leq e'$ then return True
10:            else                                               // Second iteration (see Ex. 4.1)
11:                $E'' \leftarrow$ FindMaxE($GS, dst, e'$)
12:                $e'' \leftarrow \min(b, E''[src] + w)$
13:                if $e' \leq e''$ then return True
14:                else                                           // More iterations (see Ex. 4.2)
15:                    for all states $s_M$ with $E''[s_M] = b$ do
16:                        $E_{\rightarrow} \leftarrow$ FindMaxE($GS, s_M, b$)
17:                        $e_{\text{dst}} \leftarrow \min(b, E_{\rightarrow}[src] + w)$
18:                        $E_{\leftarrow} \leftarrow$ FindMaxE($GS, dst, e_{\text{dst}}$)
19:                        if $E_{\leftarrow}[s_M] = b$ then return True
20:    return False
```

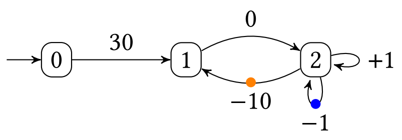
one copy of the SCC, which we call a level, per color. The first level roots the degeneralization in the original automaton; transitions leading back from the last to the first level are called back-edges. These back-edges play a crucial role as they are the only colored transitions in the degeneralized SCC and represent the accepting transitions.

Hence any Büchi accepting cycle in the degeneralization needs to contain at least one such back-edge, we can therefore focus our attention on these. We proceed to check for each back-edge whether we can embed it in a (c, b)-feasible cycle within the degeneralized SCC. This needs to be done with care and we might need multiple iterations to ensure that no such cycle exists. To this end, we start by computing the energy-optimal paths starting at the destination of the current back-edge (by construction, a state in the first level) with an initial credit corresponding to its maximal prefix energy (as found in the first step). This is done in line 7 and allows us to compute the maximal energy achievable in the destination of the back-edge when imposing the back-edge as the last transition to be taken.

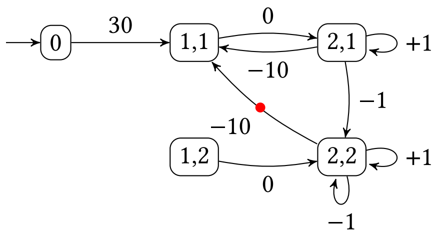
If this energy is greater than or equal to the prefix energy (in fact it can only be equal, as the prefix energy is the maximal energy attainable for this state without any additional constraints), then we can obviously traverse the same path over and over and have therefore found a (c, b)-feasible accepting lasso.

However, the converse is not true. If the computed energy for the destination of the back-edge is smaller than the prefix energy, we cannot conclude that no energy feasible cycle embedding the back-edge exists.

Example 4.1. Consider the example shown in Figure 5a. Here we have an automaton for which we have to compute maximal energy levels in the SCC twice (lines 10-13 in Algorithm 1): First we compute the prefix energy from state 0, with $b = 30$ and $c = 0$. Then we are interested in the only



(a) Original WBA



(b) Degeneralizing SCC $\{1, 2\}$ with level 1 rooted in the original WBA. Back-edges colored red.

Fig. 5. Left: WBA (also used in Example 4.1); right: degeneralization of one SCC (states named *original state*, *level*).

back-edge, leading from $(2, 2)$ to $(1, 1)$ (the states in the degeneralized SCC). The state $(1, 1)$, the destination of the back-edge, corresponds to state 1 as it is in the first level of the degeneralization, which is rooted in the original graph. The prefix energy of state 1 is 30, while its optimal energy on the cycle, after taking the back-edge, is 20. This means that despite it being part of a energy-positive loop the state has less energy than after the prefix. Hence we cannot conclude that we have found an accepting lasso after the first iteration, but need to run the algorithm once more from the state $(1, 1)$ with a new initial credit of 20 and $b = 30$. In this iteration the state $(2, 2)$ can once again reach an energy of 30, causing the new energy of state $(1, 1)$ to be 20 once more. Now we can finally conclude that a feasible lasso indeed exists.

In some cases however, even two iterations do not suffice. In fact the number of iterations necessary, when simply repeating the application of the modified Bellman-Ford with an updated initial credit, can be linear in the value of the weak upper bound b (given enough states exist) as shown in the next example

Example 4.2. Consider the WBA of Figure 6 which illustrates this problem and can be easily extended to necessitate $O(b)$ iterations.

The idea is as follows, illustrated for $b = 5$: We have a single back-edge with a weight of 0. Between the destination and source state of the back-edge (here state 1 and 6) we create $b - 1$ states with positive self-loops (here states 2 through 5). The prefix for the destination of the back-edge allows it to reach the maximal energy of b, here the prefix is simply the transition from 0 to 1 with a weight of 5. The only feasible cycle goes from state 1 to state b (here state 5) to state $b+1$ (here state 6), then finally takes the back-edge to complete the cycle. On this cycle, the energy attained by state $b + 1$ is equal to 1.

However, to find this feasible cycle, we first need to discard the cycles $1 \to i \to 6 \to 1$ for all i from 2 to $b - 1$. During the first iteration, the initial credit is equal to $b = 5$. The ideal path to the source state of the back-edge passes by state 2 allowing it to reach an energy of $b - 1 = 4$. This new initial credit for state 1 now however forbids to take the transition to state 2. The updated optimal path to the source of the back-edge now passes through state 3 and allows it to reach an energy of $b - 2 = 3$.

This continues in a similar manner for all other states up to $b - 1$ after which the actual feasible cycle is found. This example exposes a flaw in [15, Algorithm 1] which only uses two iterations to find feasible cycles and thus would fail to find this one.

Note that the above scenario cycles are hidden because they are not energy optimal but instead have a (low) constant exit energy, *i.e.,* independently of the entrance energy, the exit energy will

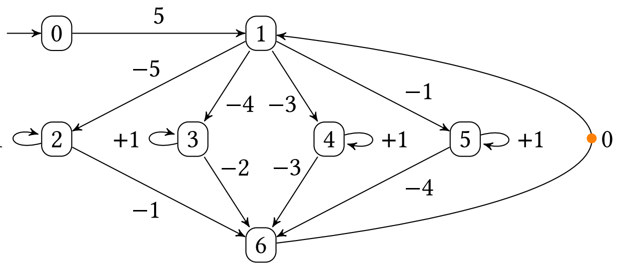


Fig. 6. WBA (of Example 4.2) where two iterations do not suffice

always be constant. In the example above, state 5 can attain energy b and after applying the exit cost of -4, we reach state 6 with an energy of 1 independently of the entrance energy at state 1 (if at least the entrance cost 1 is available). In any such scenario, the exit energy is only constant because the weak upper bound b is attained (or surpassed) along the way.

We would like to avoid having to run a number of iterations which is linear in the weak upper bound. In order to do so, we propose the following strategy: Instead of updating the initial credit, we can explicitly search for cycles which embed a state attaining maximal energy as well as the back-edge. The idea is to find this cycle by doing the following steps for every state s_M in the strongly connected component that attained maximal energy:

- Run the modified Bellman-Ford starting in s_M with initial credit b to compute the maximal energy of the source of the back-edge (line 16).
- Propagate this energy along the back-edge to find the energy of the destination (line 17)
- Run the modified Bellman-Ford starting in the destination of the back-edge using the energy computed in the last step as initial credit (line 18).

If the energy computed for s_M in the last step is once again b, that is, we can return to this state with maximal energy, then we can conclude that we have found a feasible cycle. This optimization is interesting from a practical point of view as the number of states with maximal energy is typically significantly smaller than b. Moreover it allows to bound the number of iterations necessary by the number of states rather than the value of b which is necessary to have an overall complexity which is independent of b.

Only if none of the nodes attaining maximal energy can be embedded in a positive cycle containing the current back-edge we continue with the next back-edge in the SCC or with the next SCC once all back-edges exhausted. Finally, we can conclude that no (c, b)-feasible Büchi accepting lasso exists once we have exhausted all (accepting) SCCs.

Computing the Optimal Energy

Our main Algorithm 1 allows us to decide the existence of feasible lassos in the WBA. However one of the key components, the function FindMaxE has not yet been detailed and we will do so in this section, detailing how to efficiently find the optimal energy in weighted graphs allowing positive loops.

The problem is similar (but inverse) to finding shortest paths in weighted graphs. This may be done using the well-known Bellman-Ford algorithm [4, 20], which breaks with an error if it finds negative loops. In our inverted problem, we are seeking to maximize energy, so positive loops are accepted and even desired. To take into account this particularity, we modify the Bellman-Ford algorithm to invert the weight handling and to be able to handle positive loops. The modified Bellman-Ford algorithm is given in Algorithm 2.

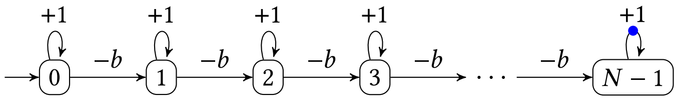


Fig. 7. WBA for Example 4.3

The standard algorithm computes shortest paths by relaxing the distance approximation until the solution is found. One round (an iteration of the outer loop over the number of states) considers all transitions to relax the respective destination node and the algorithm makes as many rounds as there are nodes. This ensures that the shortest distance is found as further improvements can only be caused by negative loops.

Inverting the algorithm is easy: the relaxation is done if the new energy is higher than the old one; additionally the new energy has to be non-negative and is bounded from above by the weak upper bound.

The second modification to Bellman-Ford is the handling of positive loops. This part is more involved, especially if one strives for an efficient algorithm. We could run Bellman-Ford until it reaches a fixed point, however this can significantly impact performance as shown in the following example.

Example 4.3. Consider the automaton shown in Figure 7 with b being a multiple of N for simplicity. Here, in order to attain the maximal energy on a state starting with 0 initial credit, $\frac{b}{N}$ iterations of the modified Bellman-Ford are necessary. This is because the self-loop increases the energy of the state by one, and the self-loop is considered N times during one iteration. The energy is however only improved for states which have already been discovered and the next state (the state to the right of the current state) can only be reached once the current state has attained b. To reach a fixed point of the energy over the entire graph we need to reach all states and then ensure that they all reach maximal energy, which is only achieved after $N\frac{b}{N} = b$ iterations of the modified Bellman-Ford.

Ideally we would like the upper bound to have no influence on the runtime. To this end we introduce the function PumpAll, which sets the energy level of all states on positive loops detected by the last iteration of Bellman-Ford to the achievable maximum. This way, instead of needing $\frac{b}{N}$ iterations of Bellman-Ford to attain the maximal energy, we only need one plus a call to PumpAll.

Before continuing, we make the following observation. This stage will be called from Algorithm 1 that recognizes loops necessary to fulfill the Büchi condition. Here, we only need to check reachability. Therefore, the only reason to form a loop is to gain energy, implying that we are only interested in *simple* energy positive loops, *i.e.,* loops where every state appears at most once. If we set the optimal reachable weight in simple loops, then nested loops are updated by Bellman-Ford in the usual way afterwards.

To improve the runtime of our algorithm, we exploit that Bellman-Ford can detect positive loops and handle these loops specifically. Note however that contrary to a statement in [7], we cannot simply set all energy levels on a positive loop to b: in the example of Figure 5a, starting in state 2 with an initial credit of 10, the energy level in state 1 will increase with every round of Bellman-Ford but never above $20 = b - 10$.

In order to have an algorithm whose complexity is independent of b, we instead compute the fixed point from above. We first make the following observation.

Lemma 4.4. *On (strictly) energy positive loops, there exists at least one state on the loop that can attain the maximal energy b.*

Algorithm 2 Modified Bellman-Ford

```
Shared Variables: $E, P$
 1: function modBF(weighted graph $G$)
 2:     for $n \in \{1, \dots, |S|\}$ do
 3:         for all $t = s \xrightarrow{w} s' \in T$ do
 4:             $e' \leftarrow \min(E(s) + w, b)$
 5:             if $E[s'] < e'$ and $e' \geq 0$ then
 6:                 $E[s'] \leftarrow e'$
 7:                 $P[s'] \leftarrow t$            // $P\colon S \to T$, mapping states to best incoming transition

    Helper function assigning optimal energy to all states on the energy positive loop containing $s$
 8: function PumpLoop(weighted graph $G$, state $s$)
 9:     for all $s' \in$ Loop(s) do                // Loop returns the states on the loop of $s$ ...
10:         $E[s'] \leftarrow -1$                  // Special value to detect fixed point
11:     $E[P[s].src] \leftarrow b$
12:     while $\top$ do                            // Loops at most twice
13:         for all $s' \in$ Loop(s) do            // ... in forward order
14:             $t \leftarrow P[s']$
15:             $e' \leftarrow \min(b, E[t.src] + t.w)$
16:             if $e' = E[t.dst]$ then
17:                 Mark loop (and suffix) as done
18:                 return                          // fixed point reached
19:             $E[t.dst] \leftarrow e'$

    Helper function, pumping all energy positive loops induced by $P$
20: function PumpAll(weighted graph $G$)
21:     for all states $s$ that changed their weight do
22:         $t = P[s]$
23:         if $\min(b, E[t.src] + t.w) > E[s]$ then
24:             $s' \leftarrow s$                  // $s$ can be either on the loop or in a suffix of one
25:             repeat                              // Go through it backwards to find a state on the loop
26:                 $s'.mark \leftarrow \top$
27:                 $s' \leftarrow t.src$
28:             until $s'$ already marked
29:             PumpLoop($G, s'$)                   // Pump it

    Function computing the optimal energy for each state
30: function FindMaxE(graph $G$, start state $s_0$, initial credit $c$)
31:     Init($s_0$, $c$)                           // initialize values in E to $-\infty$ and $E(s_0) = c$
32:     while not $fixedpoint(E)$ do               // Iteratively search for loops, then pump them
33:         modBF($G$)
34:         PumpAll($G$)
35:     return copyOf($E$)
```

Proof. Since the loop is energy positive, we can increase the energy level at any specific node by cycling through the loop. This can be repeated until a fixed point is reached. This fixed point is only reached when at one of the states the accumulated weight reaches b (or would surpass b but is then cut down to b). As the increase of energy with every loop is a strictly monotone operation, a fixed point will be reached. □

If we knew the precise state that attains maximal energy, we could set its energy to b and follow the loop once while propagating the energy, causing every state on the loop to be set to its maximal achievable energy. However, not knowing which state will effectively attain b, we start with any state on the loop, set its energy to b and propagate the energy along the loop until a fixed point is reached. This is the case after traversing the loop at most twice. This is done by the function PumpLoop.

Lemma 4.5. PumpLoop *calculates the desired fixed point after at most two iterations through the loop.*

Proof. In Algorithm 2, lines 9 and 10 ensure that the fixed point check in line 16 does not detect false positives. After setting an arbitrary state's energy to b, the algorithm iterates through the states in the loop in forward order.

Consider w.l.o.g. the positive loop $\gamma = s_1 \xrightarrow{w_1} s_2 \xrightarrow{w_2} \cdots \xrightarrow{w_{N-1}} s_N$ with $s_1 = s_N$. By Lemma 4.4 we know that there exists at least one state s_j with $0 \leq j < N$ whose maximal energy equals b. Before the first energy propagating traversal of the loop we set the energy of s_1 to b. Two cases present themselves. If $j = 0$, then energy is correctly propagated and we reach a fixed point after one traversal. In the second case, the energy attainable by s_1 is strictly smaller than b. Propagating from this energy level will over-approximate the energies reached by the states s_0 through s_{j-1} on the loop, but only until state s_j is reached which actually attains b. As energy is bounded, the energy levels of state s_j and its successors $s_{j+1}, \ldots, s_N$ are correctly calculated. This means that after traversing the loop $s_j \xrightarrow{w_j} \cdots \xrightarrow{w_{N-1}} s_N \xrightarrow{w_1} s_2 \xrightarrow{w_2} \cdots \xrightarrow{w_{j-1}} s_j$, all energy levels on the loop are correctly calculated and this is guaranteed to happen before traversing the original loop twice.

The corresponding fixed point condition is detected by line 16 which will stop the iteration. Note that we actually need to check for *changes* in the energy level on line 16, and not whether some state attained energy b, as we at this point cannot know whether this energy was reached due to over-approximation. □

Note that the pseudocode shown here is a simplification, as our implementation contains some further optimizations. Namely, we implement an early exit in modBF if we detect that a fixed point is reached, and we keep track of states which have seen an update to their energy, as this allows to perform certain operations selectively.

Algorithm complexity

We are now able to conclude our discussion from Section 2 and show that energy Büchi problems for finite WBA are decidable in polynomial time.

Proof of Theorem 2.7. For our decision procedure, the search for strongly connected components can be done in polynomial time. Our modified Bellman-Ford algorithm also has polynomial complexity. It is called once at the beginning of Algorithm 1 and then for every back-edge of every strongly connected component, it is called several times. Its amount depends on the number of energy-maximal states (line 15 of Algorithm 1). Given that the number of back-edges is bounded by the number of edges, and that the number of energy-maximal states is bounded by the number of states, we conclude that our overall algorithm has polynomial complexity. □

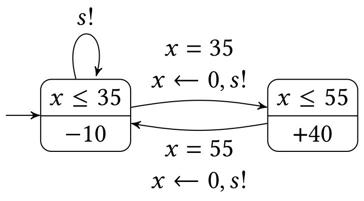


Fig. 8. Base circuit

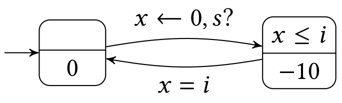


Fig. 9. Work module #i

#mod	#states	to cpa [s]	sol [s]
1	25	0.01	0.00
3	90	0.03	0.02
5	293	0.06	0.24
7	1012	0.19	3.24
9	3759	0.89	59.52
10	7377	1.87	261.38
11	14582	4.37	1194.81

Table 1. Benchmark results. From left to right: Number of work modules, Number of states in cpa, time needed to compute cpa, time needed to solve energy Büchi problem. Benchmarks done on an ASUS G14, Ryzen 4800H CPU with 16Gb RAM.

5 BENCHMARKS

We employ our running example to build a scalable benchmark case. For modeling convenience we use products of WTBAs as introduced above extended with standard sender/receiver synchronization via channels. The additional labels $s!$ and $s?$ are used for synchronization. Edges with $s!$ can always be taken and emit the signal s; edges with $s?$ can only be taken if a signal s is currently emitted. This modeling allows multiple work modules to start working at the same time.

As before, we use a base circuit with two states, see Figure 8. Work module #i, see Figure 9, uses 10 energy units while working and spends exactly i time units in the work state. We then combine these models with the specification that time must pass and that every work module is activated infinitely often. All the presented instances are schedulable. Table 1 presents the results of our benchmark, showing that the presented approach scales fairly well. We note that most of the time for solving the energy Büchi problem (last column) is spent in our Python implementation of our modified Bellman-Ford algorithm. In fact the total runtime is (at least for #mod $\geq$ 5) directly proportional to the number of times lines 4 to 7 of ModBF in Algorithm 2 are executed. Therefore, the implementation could greatly benefit from a direct integration into Spot and using its C++ engine.

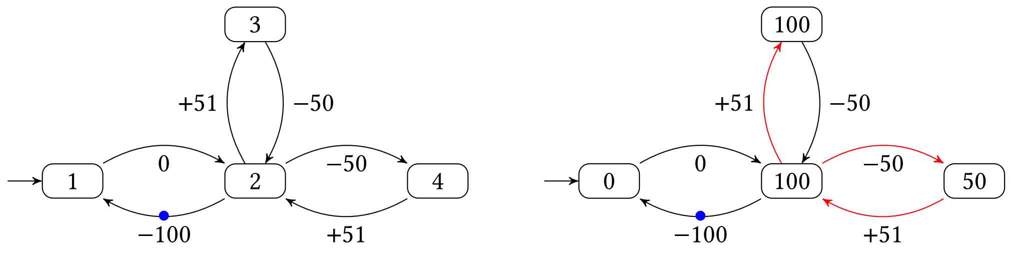
6 TRACE EXTRACTION

Our main Algorithm 1 allows us to answer the question if at least one accepting feasible lasso exists. However, the algorithm does not provide the lasso itself. In fact deducing the lasso, which is of great practical use, from the intermediate results generated by Algorithm 1 is a nontrivial task in itself. This corresponds to the energy Büchi trace problem and will be discussed in this section.

Consider our running example: affirming or refuting the existence of a feasible schedule for all work modules is of a certain interest. Extracting the actual trace which can then be used as a control strategy is however significantly more interesting.

Example 6.1. Before going into the details of the algorithm, consider the WBA given in Figure 10a with 1 being the initial state, the initial credit being set to 0 and a weak upper bound equal to 100.

In this example, all transitions need to be taken infinitely often: the transition $2 \longrightarrow 1$ is needed to satisfy the acceptance condition, it can however only be taken if the maximal amount of energy was accumulated; the loop $2 \longrightarrow 4 \longrightarrow 2$ is energy positive and allows state 2 to attain maximal



(a) Nontrivial example of trace extraction (for $b = 100$) (b) Results obtained from Algorithm 1.

Fig. 10. Example 6.1: automaton to show difficulties in trace extraction.

energy, however due to its entrance cost of 50 it cannot be taken directly after arriving in 2 from 1; in order to be able to take the loop $2 \longrightarrow 4 \longrightarrow 2$ one has first to traverse sufficiently often $2 \longrightarrow 3 \longrightarrow 2$, which has no entrance cost, but does not allow the energy in state 2 to grow beyond 50.

Ideally we would like to find the (shortest) accepted cycle which in this case is $(1 \longrightarrow (2 \longrightarrow 3 \longrightarrow 2)^{50} \longrightarrow (2 \longrightarrow 4 \longrightarrow 2)^{50} \longrightarrow 1)^{\omega}$. In this work we restrict ourselves to the easier task of finding an abstraction of the accepting cycle of the form $(1 \longrightarrow (2 \longrightarrow 3 \longrightarrow 2)^{+} \longrightarrow (2 \longrightarrow 4 \longrightarrow 2)^{+} \longrightarrow 1)^{\omega}$ where, by abuse of the usual notation, $(2 \longrightarrow 3 \longrightarrow 2)^{+}$ means that the loop $(2 \longrightarrow 3 \longrightarrow 2)$ is repeated until an energy fixed point is reached.

So why is it difficult to retrieve this trace from the results of Algorithm 1? The results are shown in Figure 10b: here states are labeled by the maximum energy for the state and the transitions leading to the energy optimal predecessor are shown in red.

This induces multiply difficulties. First, the notion of optimal predecessor only allows us to find the (energy positive) loop $(2 \longrightarrow 4 \longrightarrow 2)$ but not $(2 \longrightarrow 3 \longrightarrow 2)$. Secondly, the transition $2 \longrightarrow 1$ which is necessarily taken as it corresponds to the back-edge, is not energy optimal. Therefore, even when disregarding energy feasibility, we cannot hope to find a feasible cycle embedding the back-edge by simply following the energy optimal predecessor.

In the rest of the section we will detail an efficient trace extraction algorithm. We will show how to avoid a complete (re-)exploration of the graph and why it is necessary to slightly modify the results of Algorithm 1 to achieve this.

Adapting the solver output

The standard extension of storing the energy optimal predecessor (via the corresponding transition) in the Bellman-Ford algorithm, which corresponds to line 7 in Algorithm 2, is not enough.

The problem is that predecessors that have been energy optimal at some point during the execution of Bellman-Ford are *forgotten* once better predecessor have been found. However, these intermediate steps might be crucial to ensure energy feasibility of the path. Indeed, to avoid re-exploration of the entire graph we need to store not only the last energy optimal predecessor, but all predecessors that have been energy optimal at some point during the execution. The so-modified procedure can be found in Algorithm 3.

The implications and correctness of the optimization of not unconditionally storing all predecessors (line 7) will be discussed later on. For now, simply consider that all *necessary* predecessors are stored.

Algorithm 3 Modified Bellman-Ford All Predecessors

```
Shared Variables: $E : Array[int], P : Array[List[int]]$
1: function modBF(weighted graph $G$)
2:     for $n \in \{1, \dots, |S|\}$ do
3:         for all $t = s \xrightarrow{w} s' \in T$ do
4:             $e' \leftarrow \min(E(s) + w, b)$
5:             if $E[s'] < e'$ and $e' \geq 0$ then
6:                 $E[s'] \leftarrow e'$
7:                 if $len(P[s']) < 2$ or $P[s'][-1] \neq t$ or $P[s'][-2] \neq t$ then
8:                     $P[s'].append(t)$                    // $P\colon S \to List[T]$
```

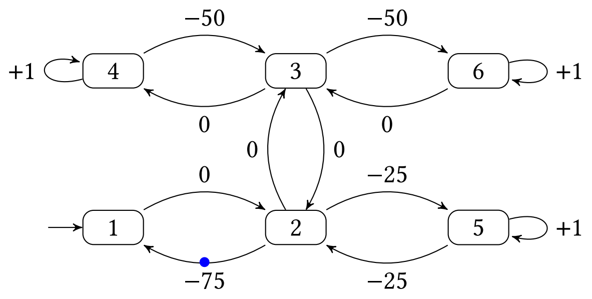


Fig. 11. Trace extraction example. An accepting cycle exists for $b = 75$.

The extraction algorithm

With the preliminaries being established, we can detail the actual trace extraction algorithm. To better motivate the algorithm we will introduce a notoriously difficult running example, highlighting most of the encountered problems.

Example 6.2. In Figure 11, every +1-self-loop allows to increase energy in that state up to the weak upper bound $b = 75$. The entrance and exist cost (the weights on the incoming and outgoing transitions) of the states 4, 5 and 6 put an implicit order on which states can be visited. State 4 is the only maximal energy state that can be directly reached from state 1 without any initial credit. As the transition from 4 to 3 has a cost of 50, we can only attain an energy of 25 in state 3, prohibiting a direct passage to state 6. To attain state 6, which will allow us to eventually traverse the back-edge $2 \longrightarrow 1$, we need to use the self-loop on state 5 as an additional positive loop.

Thus, the entrance costs limit the possibilities in which order the self-loop states may appear along the trace, *i.e.,* first in state 4, then 5 and finally 6. Therefore all feasible cycles need to have the form

$$1 \rightsquigarrow (4 \longrightarrow 4)^{\geq 75} \longrightarrow 3 \rightsquigarrow 2 \longrightarrow (5 \longrightarrow 5)^{\geq 75} \longrightarrow 2 \rightsquigarrow 3 \longrightarrow (6 \longrightarrow 6)^{\geq 75} \rightsquigarrow 2 \longrightarrow 1,$$

where $\tau^{\geq 75}$ means that the loop τ needs to be taken at least 75 times consecutively and $\rightsquigarrow$ stands for any (possibly looping) path that is at least energy-neutral.

The path segments denoted with $\rightsquigarrow$ allow for instance to take the loop $(2 \longrightarrow 3 \longrightarrow 2)$ whenever desired, as it is energy neutral. Also, in this example it is always possibly to return to positive self-loops on states with a lower state number, before continuing. For instance, the path $1 \longrightarrow 2 \longrightarrow 3 \longrightarrow (4 \longrightarrow 4)^{75} \longrightarrow 3 \longrightarrow 2 \longrightarrow (5 \longrightarrow 5)^{25} \longrightarrow 2 \longrightarrow 3 \longrightarrow (4 \longrightarrow 4)^{75} \dots$ is perfectly feasible as it can be extended to an energy-feasible accepting loop.

Algorithm 4 Trace extraction algorithm

Shared Variables:
- $E, E', E'', E_{\rightarrow}, E_{\leftarrow} : Array[int]$ Attainable Energies;
- $P, P', P'', P_{\rightarrow}, P_{\leftarrow} : Array[List[transition]]$ Extended predecessor list;
- $be : transition$ back-edge to be embedded;
- $s_M : int$ Maximal energy state to be embedded;
- $G : Graph$, $GS : Graph$ degeneralized SCC

```
1: function TraceExtraction
2:     $src, weight, dst \leftarrow be$
3:     if not $s_M$ then            // Alg. 1 exited on line 9 or 13, we search directly for a cycle
4:         $P_{cyc} \leftarrow P''$ if $P''$ else $P'$            // $P''$ and $E''$ are only set after line 10 in Alg. 1
5:         $E_{cyc} \leftarrow E''$ if $E''$ else $E'$
6:         $cyc \leftarrow$ FindPath$(GS, P_{cyc}, dst, src, E_{cyc}[dst], E_{cyc}[src], weight)$
7:         $cyc \leftarrow cyc \cdot be$            // Concatenate the path with the back-edge to form the cycle
8:         $entry \leftarrow dst$
9:     else            // Alg. 1 exited on line 19, we search for a cycle with maximal energy state $s_M$
10:        $cyc_{\rightarrow} \leftarrow$ FindPath$(GS, P_{\rightarrow}, s_M, src, b, E_{\rightarrow}[src])$
11:        $cyc_{\leftarrow} \leftarrow$ FindPath$(GS, P_{\leftarrow}, dst, s_M, min(E_{\rightarrow}[src] + weight, b), b)$
12:        $cyc \leftarrow cyc_{\leftarrow} \cdot be \cdot cyc_{\rightarrow}$            // Concatenate to form the cycle
13:        $entry \leftarrow s_M$
14:    $pref \leftarrow$ FindPath$(G, P, initialstate, entry, c, E[entry])$
15:    return $pref \cdot cyc^+$
```

Since the set of feasible cycles is infinite, we cannot consider all of them. We need an effective and principled way to investigate them that is correct and complete. As the Büchi acceptance is ensured by including the back-edge in the cycle, the rest of the path's only concern is to ensure energy feasibility. This makes it possible to further restrain the structure of paths to be considered without losing correctness of the overall algorithm.

The goal of Algorithm 4 is to split up the search for lassos into sub-paths that are easier to handle. Each of these sub-paths can be found using the corresponding energies and optimal predecessors. If it contains loops, their only purpose is to accumulate energy, otherwise they represent the optimal path from a source to a destination.

The shared variables are set in Algorithm 1 where the extended predecessor lists P, P', P'', $P_{\rightarrow}$ and $P_{\leftarrow}$ are implicitly set at the same time as their corresponding E, E', E'', $E_{\rightarrow}$ and $E_{\leftarrow}$.

The if-part of the algorithm handles the case where we only need to embed the back-edge in order to find the cycle. To this end we search for a path from the destination of the back-edge to the source which is such that the path can be closed to a feasible cycle if the back-edge has a weight of w. Since this algorithm is only called if the corresponding energy Büchi problem was feasible, we are guaranteed that such a path exists.

The else-part handles the case where the maximal energy state s_M and the back-edge need to be embedded into the cycle. To this end we first search for the optimal path from the maximal energy state s_M to the source of the back-edge, propagate the energy along the back-edge and finally search for the optimal path from the destination of the back-edge back to s_M. As before, we are guaranteed that we can find such paths and that they form a feasible cycle.

The last step is to search for the prefix of the lasso in a similar manner. Thereby the search for a lasso is split into finding several easier sub-paths with additional constraints on the initial

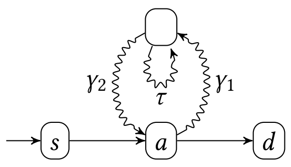


Fig. 12. Path illustrating the case of nested loops. Squiggly arrows indicate paths instead of transitions.

credit and desired accumulated energy. Before detailing the pseudocode of FindPath, let us shortly discuss its difficulties.

The backward exploration of the extended predecessor list is at its core a combinatorial problem over all the lists of possible predecessors with repeated elements. Therefore, standard graph traversal techniques fail as it might be necessary to traverse the same state multiple times. For instance in Figure 10a, state 2 appears in two different energy positive loops which are both necessary. In the following we establish several lemmas needed to break the search down to an efficient algorithm and prove its correctness.

Lemma 6.3. *Nested loops are not necessary for energy feasibility. That is, every path of the form* $s \to (a \to \gamma_1 \to (\tau)^+ \to \gamma_2 \to a)^+ \to d$ *ensuring some maximal energy* m *in* d *when starting with initial credit* c *in* s *can be decomposed into either*

(1) $s \to a \to \gamma_1 \to (\tau)^+ \to \gamma_2 \to (a \to \gamma_1 \to \gamma_2 \to a)^+ \to d$ *or*
(2) $s \to a \to \gamma_1 \to (\tau)^+ \to \gamma_2 \to a \to d$

with one of them achieving the same m *in* d *when starting with* c *in* s.

Note that the lemma is without loss of generality: for multiple nested loops it suffices to reapply the decomposition.

Proof. See Figure 12 for an illustration. The intuition for the different decompositions is as follows: In the decomposition (2), the loop τ (together with the suffix $\gamma_2 \to a$) is the energy optimal predecessor for d. Therefore it is not necessary to take the "outer" loop ($a \to \gamma_1 \to \tau \to a$) repeatedly, but γ_1 and γ_2 appear a single time on the path from s to d.

The decomposition (1) corresponds to the outer loop $a \to \gamma_1 \to \gamma_2 \to a$ being the energy optimal predecessor for state d. In this case, τ may only appear in the prefix of this loop and it might be necessary to take it repeatedly to gather enough energy before being able to take the optimal loop. For instance, imagine γ_1 to be energy neutral and γ_2 to consist of the sub-paths γ_2' and γ_2'' ($\gamma_2 = \gamma_2' \to \gamma_2''$). If γ_2' is significantly energy negative and γ_2'' is significantly energy positive, then it might be necessary to loop on τ in order to be able to complete the outer loop. Once we have gathered enough energy to traverse the outer loop once, we can always take it again, as it is energy positive overall. □

This allows us to impose an additional structure on the lassos to be considered: they all have to be of the form

$$\gamma_{p,0}\tau_{p,0}{}^+\gamma_{p,1}\tau_{p,1}{}^+\cdots\gamma_{p,k}\tau_{p,k}{}^+.(\gamma_{c,0}\tau_{c,0}{}^+\gamma_{c,1}\tau_{c,0}{}^+\cdots\gamma_{c,l}\tau_{c,l}{}^+)^\omega.$$

With $k \geq 0$ and $l > 0$, all sub-paths denoted γ correspond to loop-free segments, all sub-paths denoted with τ correspond to simple, energy positive, loops. All sub-paths denoted by γ may be empty, but the loops τ may not be empty and must be taken at least once. The subscript p (respectively c) indicates that the sub-path belongs to the prefix (respectively cycle) of the lasso.

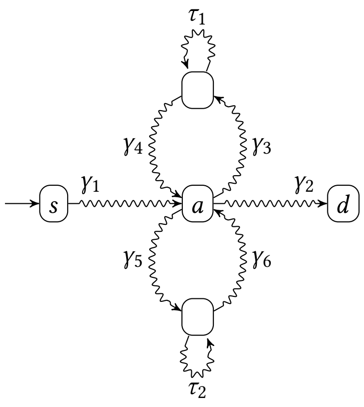


Fig. 13. Non-reappearance of positive loops.

Note that we are not interested in determining exactly how often the loops τ have to be taken, but simply assume that they are repeated until an energy fixed point is reached and have to be taken at least once, as denoted by τ^+.

Lemma 6.4. *Given two (strictly) energy positive loops τ_1 and τ_2, as shown in Figure 13, some initial credit c, an initial state s and a weak upper bound b, let m be the maximal energy attainable in d by one of the following paths:*

(1) $s \longrightarrow \gamma_1 \longrightarrow a \longrightarrow \gamma_2 \longrightarrow d$,
(2) $s \longrightarrow \gamma_1 \longrightarrow a \longrightarrow \gamma_3 \longrightarrow {\tau_1}^+ \longrightarrow \gamma_4 \longrightarrow a \longrightarrow \gamma_2 \longrightarrow d$,
(3) $s \longrightarrow \gamma_1 \longrightarrow a \longrightarrow \gamma_3 \longrightarrow {\tau_1}^+ \longrightarrow \gamma_4 \longrightarrow a \longrightarrow \gamma_5 \longrightarrow {\tau_2}^+ \longrightarrow \gamma_6 \longrightarrow a \longrightarrow \gamma_2 \longrightarrow d$,
(4) $s \longrightarrow \gamma_1 \longrightarrow a \longrightarrow \gamma_5 \longrightarrow {\tau_2}^+ \longrightarrow \gamma_6 \longrightarrow a \longrightarrow \gamma_2 \longrightarrow d$,
(5) $s \longrightarrow \gamma_1 \longrightarrow a \longrightarrow \gamma_5 \longrightarrow {\tau_2}^+ \longrightarrow \gamma_6 \longrightarrow a \longrightarrow \gamma_3 \longrightarrow {\tau_1}^+ \longrightarrow \gamma_4 \longrightarrow a \longrightarrow \gamma_2 \longrightarrow d$.

If τ_1 and τ_2 are the only (strictly) energy positive loops in the graph, then m is the best energy attainable in d when starting in s with c among all paths that can be constructed using the sub-paths $\gamma_1, \dots, \gamma_6, \tau_1, \tau_2$ (elements can be repeated).

Again, the lemma is without loss of generality: if more than two (strictly) positive loops exist, it suffices to reapply the same reasoning multiple times.

Proof. From Lemma 6.3 we already know that we do not need to consider nested loops such as $(a \longrightarrow \gamma_3 \longrightarrow {\tau_1}^+ \longrightarrow \gamma_4 \longrightarrow \gamma_5 \longrightarrow {\tau_2}^+ \longrightarrow \gamma_6 \longrightarrow a)$. This, together with the fact that following the (last / final) optimal predecessor leads either back to the initial state or a strictly positive loop leaves us with the 5 different cases mentioned above:

Case (1): The direct path $s \longrightarrow \gamma_1 \longrightarrow a \longrightarrow \gamma_2 \longrightarrow d$ is optimal, passing by τ_1 or τ_2 would not increase the energy in d, therefore neither τ_1 nor τ_2 appear multiple times.

Case (2): τ_1 is the optimal predecessor of d and the path $s \longrightarrow \gamma_1 \longrightarrow a \longrightarrow \gamma_3 \longrightarrow {\tau_1}^+ \longrightarrow \gamma_4 \longrightarrow a \longrightarrow \gamma_2 \longrightarrow d$ is (c, b)-feasible. Passing by τ_2 would not increase the energy in d and neither would passing by τ_2 and then τ_1 (it would lead to the same energy), therefore neither τ_1 nor τ_2 appear multiple times.

Case (3): As for case (1) τ_1 is the optimal predecessor of d however we need to gather energy in τ_2 to be able to take it for the first time. Once we have enough energy for τ_1 there is no reason to return to τ_2 as it is not optimal, therefore neither τ_1 nor τ_2 appear multiple times.

Case (4) and (5) are the symmetric cases for case (2) and (3) only with the roles of τ_1 and τ_2 reversed. □

From this follows that we do not need to *revisit* positive loops along a path in order to obtain the optimal energy in the destination state. We can therefore restrict the form of the feasible lassos even further, to

$$\gamma_{p,0}\tau_{p,0}{}^{+}\gamma_{p,1}\tau_{p,1}{}^{+}\cdots\gamma_{p,k}\tau_{p,k}{}^{+}.(\gamma_{c,0}\tau_{c,0}{}^{+}\gamma_{c,1}\tau_{c,0}{}^{+}\cdots\gamma_{c,l}\tau_{c,l}{}^{+})^{\omega} \tag{1}$$

with $\tau_{p,i} \neq \tau_{p,j}$ and $\tau_{c,i} \neq \tau_{c,j}$ for $i \neq j$.

The FindPath algorithm

The general idea of our FindPath algorithm is to follow the energy optimal predecessors to find a path from the source to the destination state, then evaluate whether it is feasible by forward propagating the initial credit along the path and comparing it against the minimally desired energy at the destination. However, as it was shown in the initial example, we cannot simply follow the last predecessor and expect to find a feasible trace. Nor can we examine all possible paths that can be constructed using the list of all predecessors that have been optimal at some point, as due to loops there can be infinitely many of these. We therefore need to combine the idea of searching only for practical traces with Lemmas 6.3 and 6.4 and the following notion of chronological coherence in order to construct an efficient algorithm to find trace candidates.

We say that a run ρ (possibly containing loops) $\rho = s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow a \longrightarrow (b \longrightarrow \cdots \longrightarrow c \longrightarrow b)^{+} \longrightarrow \cdots$ is *chronologically coherent* with the extended predecessors P if two conditions hold. First, for all states s' that only appear once with $s \longrightarrow s'$, s must be in $P[s']$. Secondly, for all states b that appear multiple times on ρ, we first create a list $Pa[b]$ of *actual predecessors.* To this end we traverse ρ, and each time we encounter the state b with some $x \longrightarrow b$, we append x to $Pa[b]$. Now ρ is chronologically coherent with P if an index array Idx exists that is monotone and satisfies that for all $i < len(Pa[b])$, $Pa[b]_i = P[b]_{Idx[i]}$. Hence the extended predecessor list P contains all predecessors of any b, in the correct order, and consecutive repetitions may be reduced to a single occurence.

Lemma 6.5. *Given an initial state s and $c, b \in \mathbb{N}$, let E and P be the attainable energies and the extended predecessor list resulting from the call to* FindMaxE(G, s, c). *Then for every state s' there exists a run $s \longrightarrow \cdots \longrightarrow s'$ that attains the maximal achievable energy $E[s']$ and which is chronologically coherent with P.*

Proof. Since we traverse every loop on a run until an energy fixed point is reached, storing the same predecessor at most twice consecutively is sufficient. One entry ensures that a positive loop can be found, the second entry ensures that we can retake a loop partially in case the entry-point and the exit point of the loop do not coincide.

Otherwise chronological coherence is ensured by construction as we store all predecessors. □

Ensuring feasibility

Due to the asymmetry of the weak upper bound we cannot simply compute the correct energy levels via back propagation, even when following predecessors that have been energy optimal at some point. We need to use an alternation between backward search and forward exploration. We first construct a path, which might contain loops, from the destination state to the source state by following one of the optimal predecessors at each step. Once such a path is found, we need to check its energy feasibility using forward exploration (along the path).

Algorithm 5 FindPath algorithm

Shared Variables:

- $P : Array[List[transition]]$ Extended predecessor list;
- $gSrc : int$ and $gDst : int$ Initial and final state of the trace;
- $cSrc : int$ Initial energy at $gSrc$;
- $eDst : int$ Minimal energy to attain at $gDst$;
- $ext : int \cup None$ Extra cost for loop completion; $None$ if simple path

```
1:  function FindPath($G$, $P$, $gSrc$, $gDst$, $cSrc$, $eDst$, $ext = None$)
2:      /* Search for a trace starting in gSrc with cSrc energy to gDst with at least eDst energy */
3:      /* ext serves as an indicator if we search for an implicit cycle. */
4:      $ci \leftarrow [\text{len}(P[s])$ for $s$ in $range(G.numStates())]$
5:      return BackwardsSearch($ci$, [])            // Returned list is empty if no traces exists

6:  function BackwardsSearch($ci : Array[int]$ current index, $p : List[transition]$ current
    trace)
7:      $vCurr \leftarrow p.\text{front}().src$ if not $p.\text{empty}()$ else $gDst$
8:      if $vCurr = gSrc$ then                       // We found the target
9:          if ForwardExp($p$) then return $p$        // A feasible trace was found
10:     for all $i$ from $ci[vCurr] - 1$ to 0 do   // The index array ci ensures chronological coherence
11:         $ci' \leftarrow ci.\text{copy}()$
12:         $p' \leftarrow p.\text{copy}()$
13:         $ci'[vCurr] \leftarrow i$
14:         $p'.\text{pushFront}(P[vCurr][i])$       // Add the transition to the start of the trace
15:         $tr \leftarrow$ BackwardsSearch($ci', p'$)
16:         if $tr \neq []$ then return $p'$
17:     return []                                  // All options exhausted; No feasible trace.

18: function ForwardExp($p$ candidate trace)
19:     /* The extra cost ext is set to None if we do not search for a cycle */
20:     $tc \leftarrow$ CompressTrace($p$)           // Decompose trace into loops and prefixes
21:     $e \leftarrow cSrc$                          // Initial energy is defaulted to initial credit
22:     $eTarget \leftarrow eDst$
23:     for all _ in $range(1 + (ext$ is $None))$ do  // Loop twice if ext is given
24:         for all $pref, cyc$ in $tc$ do
25:             $succ, e \leftarrow \text{propAlong}(e, pref)$   // Propagate energy along prefix
26:             if not $succ$ then return False      // Prefix was not energy feasible
27:             $succ, e \leftarrow \text{tryPumpLoop}(e, cyc)$ // Detects whether loop is energy positive and feasi-
                ble
28:             if not $succ$ then return False // Loop was not energy feasible or not energy positive
29:         if $e \geq eTarget$ then return True   // Check whether enough energy was accumulated
30:         $e \leftarrow min(e + ext, b)$              // Close the implicit cycle
31:         $eTarget \leftarrow e$
32:     return False
```

The forward exploration can be done in linear time in the size of the WBA. The backward search is more complicated and is only guaranteed to terminate thanks to the lemmas above. The FindPath function in Algorithm 5 starts a backward search over all chronologically coherent traces. It switches to a forward exploration once a candidate is found, in order to evaluate its energy feasibility. In Algorithm 5, the parameters given to FindPath are afterwards shared between the called functions.

Algorithm 6 shows some helper functions in order to do the forward exploration efficiently. TraceCompression turns a list of transitions into a list of path segments by collecting and collapsing sequences of concatenable transitions. A path segment is a pair of a linear trace and a loop part. A list of path segments build a trace in the sense of Equation (1). PropAlong is a simple procedure which propagates energy along a path (or returns *False* if energy drops below 0). tryPumpLoop tries to increase energy along a loop: it returns *False* if the loop is infeasible or energy non-positive; otherwise it computes the maximal energy fixed point from above.

Example 6.6. We continue the preceding Example 6.2 with Figure 11. Invoking FindMaxE on it with initial credit 0 and a weak upper bound 75 will return $[1, 3, 5, 3]$ as extended predecessor list of state 2 (the most interesting state in this example).

This predecessor list is chronologically coherent with the trace

$$\begin{aligned}\tau = \big(1 &\to 2 \to 3 \to (4 \to 4)^{75}\\ &\to 3 \to 2 \to (5 \to 5)^{75}\\ &\to 2 \to 3 \to (6 \to 6)^{75} \to 3 \to 2 \to 1\big)^{+}\end{aligned}$$

The trace found by our algorithm is

$$\begin{aligned}\tau = \big(1 &\to 2 \to 3 \to (4 \to 4)^{+}\\ &\to 3 \to 2 \to (5 \to 5)^{+}\\ &\to 2 \to 3 \to (6 \to 6)^{+} \to 3 \to 2 \to 1\big)^{+}\end{aligned}$$

which is the expected result given that we do not seek to identify how often loops have to be taken but always assume retaking them until an energy fixed point is reached.

Note that the extended predecessor list of a state can depend on the number of states in the automaton. For instance, if Figure 11 would only show a part of the automaton and there were 50 additional states, then the extended predescessor list of 2 would become $[1, 3, 3, 5, 5, 3, 3]$. This is due to the additional iterations in the modified Bellman-Ford algorithm, allowing the states 3 and 5 to reach higher energy levels which will in turn propagate to state 2.

7 PARITY CONDITION

In this final section we show how to adapt our solution to Parity automata. Let $(\mathcal{M}, S, s_0, T)$ be a WBA, but now $p : \mathcal{M} \to \mathbb{N}$ is a function assigning non-negative integers (*i.e.,* priorities) to colors. An infinite run $\rho = s_1 \to_{M_1} s_2 \to_{M_2} \cdots$ is *Parity accepted* if the maximal priority seen infinitely often along ρ is even, that is, if $\max\{p(m) \mid m \in \mathrm{Inf}((M_i)_{i\geq 1})\}$ is even. As the number of colors is finite, the maximum always exists.

An example with priorities up to 4 is shown symbolically in Figure 14 with accepting regions represented as white zones, whereas rejecting ones are colored gray. A run ρ is accepted if either priority 4 occurs infinitely often, or priority 3 occurs finitely often and priority 2 occurs infinitely often, or priority 1 occurs finitely often and priority 0 occurs infinitely often.

In order to solve energy Parity problems, we transform them to successive energy Büchi problems. We calculate prefix energies on the original automaton, as we did in the Büchi case. For

Algorithm 6 FindPath algorithm – Helper functions

Shared Variables:

- $P : Array[List[transition]]$ Extended predecessor list;
- $be : transition$ Back-edge of interest;
- $c : int$ Initial energy at $be.src$;
- $G : Graph$

```
    Transforms a trace given as a list of transitions into a list of path segments
 1: function TraceCompression($p : List[transition]$ the trace to compress)
 2:     $idx \leftarrow 0$
 3:     $tc \leftarrow []$                                          // List containing the path segments
 4:     while $idx < $ len$(p)$ do                                 // Make sure to treat the entire list
 5:         $subtrace \leftarrow []$
 6:         $srcIdx \leftarrow Dict()$                  // Look-up mapping src vertex to index in subtrace
 7:         while $idx < $ len$(p)$ do                               // Search for the next path segment
 8:             $subtrace$.pushBack$(p[idx])$               // Add the transition to the end of the trace
 9:             $idx \leftarrow idx + 1$
10:             if $subtrace$.back$().dst$ in $srcIdx.keys()$ then                       // We found a loop
11:                 $cutIdx \leftarrow srcIdx[subTrace$.back$().dst]$
12:                 $tc$.pushBack(pathSegment($subtrace[: cutIdx], subtrace[cutIdx :]$))
13:                 break                                                   // Advance to next segment
14:             $srcIdx[subtrace$.back$().src] \leftarrow $ len$(subtrace) - 1$
15:     if $subtrace$ then                                           // add remaining subtraces to tc
16:         $tc$.pushBack(pathSegment($subtrace$, []))
17:     return $tc$

    Propagate Energy along a path
18: function PropAlong($e : int$ the energy before, $p : List[transitions]$: the path)
19:     for all $(s, weight, t)$ in $p$ do
20:         $e' \leftarrow min(b, e + weight)$
21:         if $e' < 0'$ then return False, 0
22:         $e \leftarrow e'$
23:     return True, $e$

    Pump a possibly positive loop
24: function tryPumpLoop($e : int$ the energy before, $p : List[transitions]$: a cycle)
25:     if not $p$ then
26:         return True, $e$
27:     $eInit \leftarrow e$
28:     $succ, e \leftarrow$ PropAlong$(e, p)$        // Take the loop once to determine if positive and necessary
29:     if (not $succ$) or $e \leq eInit$ then return False, 0
30:     $e \leftarrow b$                            // Set energy to the upper bound and correct via propagation
31:     $\_, e \leftarrow$ PropAlong$(e, p)$
32:     $\_, e \leftarrow$ PropAlong$(e, p)$
33:     return True, $e$
```

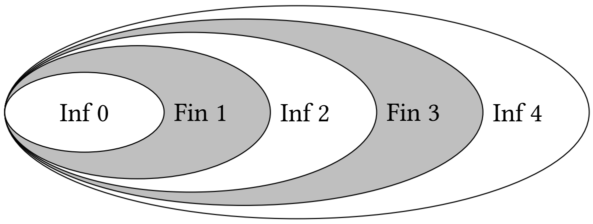


Fig. 14. Parity condition for priorities up to 4:
Inf(4) | (Fin(3) & (Inf(2) | (Fin(1) & Inf(0))))

every SCC found by Couvreur's algorithm that contains at least one transition of even priority (in SCCs containing only odd priorities, all cycles are rejected), we do the following: First we create a new automaton that is a copy of the current SCC. Then we give it to an algorithm performing the following steps.

- If the automaton is empty, *i.e.,* it has no transitions, answer negatively.
- If the highest priority is even:
  - create a copy of the current automaton;
  - set the acceptance condition to Büchi;
  - recolor the automaton the following way: all transitions with highest priority become Büchi accepted, all others are uncolored;
  - solve the energy Büchi problem for this automaton. If the result is positive, we answer positively as well. If not, then we remove all transition with the highest priority from the automaton and perform a recursive call.
- If the highest priority is odd:
  - remove all transitions with the highest priority from the automaton;
  - perform a recursive call.

This shows that the energy Parity problem can be reduced into several energy Büchi problems. Moreover, this allows us to use the same algorithm for trace extraction in the parity case.

8 CONCLUSION

We have shown how to efficiently solve energy Büchi problems, both in finite weighted (transition-based generalized) Büchi automata and in one-clock weighted timed Büchi automata, as well as how to efficiently extract the actual trace from the intermediate results. We have also extended our results to Parity conditions.

We have implemented all our algorithms in a tool based on TChecker and Spot. Solving the latter problem is done by using the corner-point abstraction to translate the weighted timed Büchi automaton to a finite weighted Büchi automaton; the former problem is handled by interleaving a modified version of the Bellman-Ford algorithm with Couvreur's algorithm.

Our tool is able to handle some interesting examples, but the restriction to one-clock weighted timed Büchi automata without weights on edges does impose some constraints on modeling. We believe that trying to lift the one-clock restriction is unrealistic; but weighted edges (without Büchi conditions) have been treated in [6], and we suspect that their approach should also be feasible here. (See [10] for related work.)

As a last remark, it is known that multiple clocks, multiple weight dimensions, and even turning the weak upper bound into a strict one which may not be exceeded, rapidly leads to undecidability results, see [7, 8, 18, 28], and we are wondering whether some of these may be sharpened when using Büchi or Parity conditions.

ACKNOWLEDGMENTS

We are grateful to Rania Saadi for her help in implementing and testing our trace extraction algorithm and the fruitful discussions during her internship. The first author Sven Dziadek is supported by the ANR project EQUUS, grant number ANR-19-CE48-0019, and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grant number 431183758.

REFERENCES

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

[2] Giovanni Bacci, Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, and Pierre-Alain Reynier. Optimal and robust controller synthesis using energy timed automata with uncertainty. *Formal Aspects of Computing*, 33(1):3–25, 2021.

[3] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *HSCC*, volume 2034 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2001.

[4] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[5] Morten Bisgaard, David Gerhardt, Holger Hermanns, Jan Krčál, Gilles Nies, and Marvin Stenger. Battery-aware scheduling in low orbit: The GomX-3 case. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM*, volume 9995 of *Lecture Notes in Computer Science*, pages 559–576. Springer, 2016.

[6] Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, and Nicolas Markey. Timed automata with observers under energy constraints. In Karl Henrik Johansson and Wang Yi, editors, *HSCC*, pages 61–70. ACM, 2010.

[7] Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, and Jiří Srba. Infinite runs in weighted timed automata with energy constraints. In Franck Cassez and Claude Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008.

[8] Patricia Bouyer, Kim G. Larsen, and Nicolas Markey. Lower-bound-constrained runs in weighted timed automata. *Performance Evaluation*, 73:91–109, 2014.

[9] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1–11. Elsevier, 1966.

[10] David Cachera, Uli Fahrenberg, and Axel Legay. An ω-algebra for real-time energy problems. *Logical Methods in Computer Science*, 15(2), 2019.

[11] Krishnendu Chatterjee and Laurent Doyen. Energy parity games. *Theoretical Computer Science*, 458:49–60, 2012.

[12] Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Generalized mean-payoff and energy games. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 8 of *Leibniz International Proceedings in Informatics*, pages 505–516, 2010.

[13] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer, 1999.

[14] Alexandre Duret-Lutz, Étienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. From Spot 2.0 to Spot 2.10: What's new? In *CAV*, volume 13372 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2022.

[15] Sven Dziadek, Uli Fahrenberg, and Philipp Schlehuber-Caissier. Energy Büchi problems. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *FM*, volume 14000 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2023.

[16] Zoltán Ésik, Uli Fahrenberg, Axel Legay, and Karin Quaas. An algebraic approach to energy problems I: *-continuous Kleene ω-algebras. *Acta Cybyernetica*, 23(1):203–228, 2017.

[17] Zoltán Ésik, Uli Fahrenberg, Axel Legay, and Karin Quaas. An algebraic approach to energy problems II: The algebra of energy functions. *Acta Cybyernetica*, 23(1):229–268, 2017.

[18] Uli Fahrenberg, Line Juhl, Kim G. Larsen, and Jiří Srba. Energy games in multiweighted automata. In Antonio Cerone and Pekka Pihlajasaari, editors, *ICTAC*, volume 6916 of *Lecture Notes in Computer Science*, pages 95–115. Springer, 2011.

[19] Heiko Falk, Kevin Hammond, Kim G. Larsen, Björn Lisper, and Stefan M. Petters. Code-level timing analysis of embedded software. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *EMSOFT*, pages 163–164. ACM, 2012.

[20] Lester R. Ford. *Network Flow Theory.* RAND Corporation, Santa Monica, CA, 1956.

[21] Goran Frehse, Kim G. Larsen, Marius Mikučionis, and Brian Nielsen. Monitoring dynamical signals while testing timed aspects of a system. In Burkhart Wolff and Fatiha Zaïdi, editors, *ICTSS*, volume 7019 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2011.

[22] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[23] Frédéric Herbreteau and B. Srivathsan. Coarse abstractions make Zeno behaviours difficult to detect. *Logical Methods in Computer Science*, 9(1), 2011.

[24] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. Better abstractions for timed automata. *Information and Computation*, 251:67–90, 2016.

[25] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Model checking timed automata with one or two clocks. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2004.

[26] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[27] Marius Mikučionis, Kim G. Larsen, Jacob Illum Rasmussen, Brian Nielsen, Arne Skou, Steen Ulrik Palm, Jan Storbank Pedersen, and Poul Hougaard. Schedulability analysis using Uppaal: Herschel-Planck case study. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (2)*, volume 6416 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2010.

[28] Karin Quaas. On the interval-bound problem for weighted timed automata. In Adrian Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *LATA*, volume 6638 of *Lecture Notes in Computer Science*, pages 452–464. Springer, 2011.

[29] Florian Renkin, Alexandre Duret-Lutz, and Adrien Pommellet. Practical "paritizing" of Emerson–Lei automata. In *ATVA*, volume 12302 of *Lecture Notes in Computer Science*, pages 127–143. Springer, October 2020.

[30] Yaron Velner, Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, Alexander Moshe Rabinovich, and Jean-François Raskin. The complexity of multi-mean-payoff and multi-energy games. *Information and Computation*, 241:177–196, 2015.

[31] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *FOCS*, pages 185–194. IEEE Computer Society, 1983.

[32] Wiesław Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1):135–183, 1998.