Acacia-Bonsai: A Modern Implementation of Downset-Based LTL Realizability

Michaël Cadilhac[1](✉) and Guillermo A. Pérez[2]

[1] DePaul University, Chicago, USA
michael@cadilhac.name
[2] University of Antwerp – Flanders Make, Antwerp, Belgium
guillermo.perez@uantwerp.be

Abstract. We describe our implementation of downset-manipulating algorithms used to solve the realizability problem for linear temporal logic (LTL). These algorithms were introduced by Filiot et al. in the 2010s and implemented in the tools Acacia and Acacia+ in C and Python. We identify degrees of freedom in the original algorithms and provide a complete rewriting of Acacia in C++20 articulated around genericity and leveraging modern techniques for better performance. These techniques include compile-time specialization of the algorithms, the use of SIMD registers to store vectors, and several preprocessing steps, some relying on efficient Binary Decision Diagram (BDD) libraries. We also explore different data structures to store downsets. The resulting tool is competitive against comparable modern tools.

Keywords: LTL synthesis · C++ · downset · antichains · SIMD · BDD

1 Introduction

Nowadays, hardware and software systems are everywhere around us. One way to ensure their correct functioning is to automatically synthesize them from a formal specification. This has two advantages over alternatives such as testing and model checking: the design part of the program-development process can be completely bypassed and the synthesized program is correct by construction.

In this work we are interested in synthesizing *reactive systems* [17]. These maintain a continuous interaction with their environment. Examples of reactive systems include communication, network, and multimedia protocols as well as operating systems. For the specification, we consider *linear temporal logic* (LTL) [27]. LTL allows to naturally specify time dependence among events that make up the formal specification of a system. The popularity of LTL as a formal specification language extends to, amongst others, AI [15,8,16], hybrid systems and control [6], software engineering [21], and bio-informatics [1].

The classical doubly-exponential-time synthesis algorithm can be decomposed into three steps: 1. *compile* the LTL formula into an automaton of exponential size [32], 2. *determinize* the automaton [29,26] incurring a second exponential blowup, and 3. determine the winner of a *two-player zero-sum game*

© The Author(s) 2023
S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 192–207, 2023.
https://doi.org/10.1007/978-3-031-30820-8_14

played on the latter automaton [28]. Most alternative approaches focus on avoiding the determinization step of the algorithm. This has motivated the development of so-called Safra-less approaches, e.g., [20,11,10,31]. Worth mentioning are the on-the-fly game construction implemented in the Strix tool [24] and the *downset*-based (or "antichain-based") on-the-fly bounded determinization described in [13] and implemented in Acacia+ [5]. Both avoid constructing the doubly-exponential deterministic automaton. Acacia+ was not ranked in recent editions of SYNTCOMP [18] (see `http://www.syntcomp.org/`) since it is no longer maintained despite remaining one of the main references for new advancements in the field (see, e.g., [12,33,30,22,2]).

Contribution. We present the Acacia approach to solving the problem at hand and propose a new implementation that allows for a variety of optimization steps. For now, we have focused on *(Büchi automata) realizability*, i.e., the decision problem which takes as input an automaton compiled from the LTL formula and asks whether a controller satisfying it exists. In our tool, we compile the input LTL formula into an automaton using Spot [9]. We entirely specialize our presentation on the technical problem at hand and strive to distillate the algorithmic essence of the Acacia approach in that context. The main algorithm is presented in Section 3.4 and the different implementation options are listed in Section 4. Benchmarks are included in Section 6.

All benchmarks were executed on the revision of the software that can be found at: `https://github.com/gaperez64/acacia-bonsai/tree/SYNTCOMP22`.

2 Preliminaries

Throughout this paper, we assume the existence of two alphabets, I and O; although these stand for input and output, the actual definitions of these two terms is slightly more complex: An *input* (resp. *output*) is a boolean combination of symbols of I (resp. O) and it is *pure* if it is a *conjunction* in which *all* the symbols in I (resp. O) appear exactly once; e.g., with $I = \{i_1, i_2\}$, the expressions $\top$ (true), $\bot$ (false), and $(i_1 \vee i_2)$ are inputs, and $(i_1 \wedge \neg i_2)$ is a pure input. Similarly, an *IO* is a boolean combination of symbols of $I \cup O$, and it is *pure* if it is a conjunction in which all the symbols in $I \cup O$ appear exactly once. We use i, j to denote inputs and x, y for IOs. Two IOs x and y are *compatible* if $x \wedge y \neq \bot$.

A *Büchi automaton* $\mathcal{A}$ is a tuple (Q, q_0, δ, B) with Q a set of states, q_0 the initial state, δ the transition relation that uses IOs as labels, and $B \subseteq Q$ the set of Büchi states. The actual semantics of this automaton will not be relevant to our exposition, we simply note that these automata are usually defined to recognize infinite sequences of pure IOs. We assume, throughout this paper, the existence of some automaton $\mathcal{A}$.

We will be interested in valuations of the states of $\mathcal{A}$ that encode the number of visits to Büchi states—again, we do not go into details here. We will simply speak of *vectors over* $\mathcal{A}$ for elements in $\mathbb{Z}^Q$, mapping states to integers. We

will write $\vec{v}$ for such vectors, and v_q for its value for state q. In practice, these vectors will range into a finite subset of $\mathbb{Z}$, with -1 as an implicit minimum value (meaning that $(-1) - 1$ is still -1) and an upper bound provided by the problem.

For a vector $\vec{v}$ over $\mathcal{A}$ and an IO x, we define a function that takes one step back in the automaton, decreasing components that have seen Büchi states. Write $\chi_B(q)$ for the function mapping a state q to 1 if $q \in B$, and 0 otherwise. We then define $\text{bwd}(\vec{v}, x)$ as the vector over $\mathcal{A}$ that maps each state $p \in Q$ to:

$$\min_{\substack{(p,y,q)\in\delta \\ x \text{ compatible with } y}} (v_q - \chi_B(q)) \quad ,$$

and we generalize this to sets: $\text{bwd}(S, x) = \{\text{bwd}(\vec{v}, x) \mid \vec{v} \in S\}$. For a set S of vectors over $\mathcal{A}$ and a (possibly nonpure) input i, define:

$$\text{CPre}_i(S) = S \cap \bigcup_{\substack{x \text{ pure IO} \\ x \text{ compatible with } i}} \text{bwd}(S, x) \quad .$$

It can be proved that iterating CPre with any possible pure input stabilizes to a fixed point that is independent from the order in which the inputs are selected. We define $\text{CPre}^*(S)$ to be that set.

All the sets that we manipulate will be *downsets*: we say that a vector $\vec{u}$ dominates another vector $\vec{v}$ if for all $q \in Q$, $u_q \geq v_q$, and we say that a set is a downset if $\vec{u} \in S$ and $\vec{u}$ dominates $\vec{v}$ implies that $\vec{v} \in S$. This allows to implement these sets by keeping only dominating elements, which form, as they are pairwise nondominating, an *antichain*. In practice, it may be interesting to keep more elements than just the dominating ones or even to keep all of the elements to avoid the cost of computing domination.

Finally, we define Safe_k as the downset $\{i \mid i \leq k\}^Q$, i.e., all vectors with values bounded by k. We are now equipped to define the computational problem we focus on:

BackwardRealizability
- **Given:** A Büchi automaton $\mathcal{A}$ and an integer $k > 0$,
- **Question:** Is there a $\vec{v} \in \text{CPre}^*(\text{Safe}_k)$ with $v_{q_0} \geq 0$?

We note, for completeness, that (for sufficiently large values of k) this problem is equivalent to deciding the realizability problem associated with $\mathcal{A}$: the question has a positive answer if and only if the *output player* wins the Gale-Stewart game with payoff set the *complement* of the language of $\mathcal{A}$.

3 Realizability algorithm

The problem admits a natural algorithmic solution: start with the initial set, pick an input i, apply CPre_i on the set, and iterate until all inputs induce no change to the set, then check whether this set contains a vector that maps q_0 to 0. We first introduce some degrees of freedom in this approach, then present a slight twist on that solution that will serve as a canvas for the different optimizations.

3.1 Boolean states

This opportunity for optimization was identified in [4] and implemented in Acacia+, we simply introduce it in a more general setting and succinctly present the original idea when we mention how it can be implemented in Section 4.2. We start with an example. Consider the Büchi automaton from Figure 1 with $q_0, q_1 \notin B$.

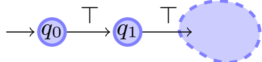

Fig. 1. Small automaton with $q_0, q_1 \notin B$.

Recall that we are interested in whether the initial state can carry a non-negative value, after CPre has stabilized. In that sense, the crucial information associated with q_0 is boolean in nature: is its value positive or -1? Even further, this same remark can be applied to q_1 since q_1 being valued 6 or 7 is not important to the valuation of q_0. Hence the set of states may be partitioned into integer-valued states and boolean-valued ones. Naturally, detecting which states can be made boolean comes at a cost and not doing it is a valid option.

3.2 Actions

For each IO x, we will have to compute $\text{bwd}(\vec{v}, x)$ oftentimes. This requires to refer to the underlying Büchi automaton and checking for each transition therein whether x is compatible with the condition. It may be preferable to precompute, for each x, what are the relevant pairs (p, q) for which x can go from p to q. We call the set of such pairs the *io-action* of x and denote it $\text{io-act}(x)$; in symbols:

$$\text{io-act}(x) = \{(p, q) \mid (\exists (p, y, q) \in \delta)[x \text{ is compatible with } y]\} \ .$$

Further, as we will be computing $\text{CPre}_i(S)$ for inputs i, we abstract in a similar way the information required for this computation. We use the term *input-action* for the set of io-actions of IOs compatible with i and denote it $\text{i-act}(i)$; in symbols:

$$\text{i-act}(i) = \bigcup_{\substack{x \text{ an IO} \\ \text{compatible with } i}} \text{io-act}(x) \ .$$

In other words, actions contain exactly the information necessary to compute CPre. Note that from an implementation point of view, we do not require that the actions be precomputed. Indeed, when iterating through pairs $(p, q) \in \text{io-act}(x)$, the underlying implementation can choose to go back to the automaton.

3.3 Sufficient inputs

As we consider the transitions of the Büchi automaton as being labeled by boolean expressions, it becomes more apparent that some pure IOs can be redundant. For instance, consider a Büchi automaton with $I = \{i\}, O = \{o_1, o_2\}$, but the only transitions compatible with i are labeled $(i \wedge o_1)$ and $(i \wedge \neg o_1)$. Pure IOs compatible with the first label will be $(i \wedge o_1 \wedge o_2)$ and $(i \wedge o_1 \wedge \neg o_2)$, but certainly, these two IOs have the same io-actions, and optimally, we would only consider $(i \wedge o_1)$. However, we should not consider $(i \wedge o_2)$, as it induces an io-action that is not induced by a pure IO. We will thus allow our main algorithm to select certain inputs and IOs and introduce the following notion:

Definition 1. *An IO (resp. input) is* valid *if there exists any pure IO (resp. input) with the same io-action (resp. input-action). A set X of valid IOs is* sufficient *if it represents all the possible io-actions of pure IOs:* $\{\text{io-act}(x) \mid x \in X\} = \{\text{io-act}(x) \mid x \text{ is a pure IO}\}$. *A sufficient set of inputs is defined similarly with input-actions.*

3.4 Algorithm

We solve **BackwardRealizability** by computing CPre^* explicitly:

Algorithm 1 Main algorithm

Input: A Büchi automaton $\mathcal{A}$, an integer $k > 0$
Output: Whether $(\exists \vec{v} \in \text{CPre}^*(\text{Safe}_k))[v_{q_0} \geq 0]$

1. Possibly remove some useless states in $\mathcal{A}$
2. Split states of $\mathcal{A}$ into boolean and nonboolean
3. Let `Downset` be a type for downsets using a vector type that possibly has a boolean part
4. Let $S = \text{Safe}_k$ of type `Downset`
5. Compute a sufficient set E of inputs
6. Compute the input-actions of E
7. **while** true **do**
8.   Pick an input-action a of E
9.   **if** no action is returned **then**
10.   **return** whether a vector in S maps q_0 to a nonnegative value
11. $S \leftarrow \text{CPre}_a(S)$

Our algorithm requires that the "input-action picker" used in line 8 decides whether we have reached a fixed point. As the picker could check whether S has changed, this is without loss of generality.

The computation of CPre_a is the intuitive one, optimizations therein coming from the internal representation of actions. That is, it is implemented by iterating through all io-actions compatible with a, applying bwd on S for each of them, taking the union over all these applications, and finally intersecting the result with S.

4 The many options at every line

The main computational costs of the algorithm are in finding input-actions and computing CPre_a. For the former, reducing the number of candidates is crucial (by considering a good set of sufficient inputs). For the latter, reducing the size of the automaton (hence the dimension of the vectors) and providing efficient data types for downsets is key. Additionally, for the "input-action picker" to return an input that *will* make progress, it has to explore S in some way — this can again be a costly operation that would be sped up by better data structures for downsets. Let us now review these potential optimizations line by line.

4.1 Preprocessing of the automaton (line 1)

In this step, one can provide a heuristic that removes certain states that do not contribute to the computation. We provide an optional step that detects *surely losing states*, as presented in [14].

4.2 Boolean states (line 2)

We provide an implementation of the detection of boolean states, in addition to an option to not detect them. Our implementation is based on the concept of *bounded state*, as presented in [4]. A state is *bounded* if it cannot be reached from a Büchi state that lies in a nontrivial strongly connected component. This can be detected in several ways, although it is not an intrinsically costly operation.

4.3 Vectors and downsets (line 3)

The most basic data structure in the main algorithm is that of a vector used to give a value to the states. We provide a handful of different vector classes:

- Standard C++ vector and array types (`std::vector`, `std::array`). Note that arrays are of fixed size; our implementation precompiles arrays of different sizes (up to 300 by default), and defaults to vectors if more entries are needed.
- Vectors and arrays backed by SIMD[3] registers. This makes use of the type `std::experimental::simd` and leverages modern CPU optimizations.

Additionally, all these implementations can be glued to an array of booleans (`std::bitset`) to provide a type that combines boolean and integer values. These types can optionally expose an integer that is compatible with the partial order (here, the sum of all the elements in the vector: if $\vec{u}$ dominates $\vec{v}$, then the sum of the elements in $\vec{u}$ is larger than that of $\vec{v}$). This value can help the downset implementations in sorting the vectors.

Downset types are built on top of a vector type. We provide:

[3] SIMD: Single Instruction Multiple Data, a set of CPU instructions & registers to compute component-wise operations on fixed-size vectors.

- Implementations using sets or vectors of vectors, either containing only the dominating vectors, or containing explicitly all the vectors;
- An implementation that relies on k-d trees, a space-partitioning data structure for organizing points in a k-dimensional space; [3]
- Implementations that store the vectors in specific bins depending on the information exposed by the vector type.

4.4 Selecting sufficient inputs (line 5)

Recall our discussion on sufficient inputs of Section 3.3. We introduce the notion of *terminal* IO following the intuition that there is no restriction of the IO that would lead to a more specific action:

Definition 2. *An IO x is said to be* terminal *if for every compatible IO y, we have* $\text{io-act}(x) \subseteq \text{io-act}(y)$*. An* input *$i$ is said to be* terminal *if for every compatible input j we have* $\text{i-act}(i) \subseteq \text{i-act}(j)$*.*

Our approaches to input selection focus on efficiently searching for a sufficient set of terminal IOs and inputs. The key property of terminal inputs is that they are automatically valid, while still being more general than pure inputs.

Proposition 1. *Any pure IO and any input is terminal. Any terminal IO and any terminal input is valid.*

Proof. Any pure IO is terminal. Consider a pure IO x and a compatible IO y. If $(p, q) \in \text{io-act}(x)$, then there is a transition $(p, z, q) \in \delta$ such that x is compatible with z, and thus $x \wedge z = x$. Consequently, $x \wedge z \wedge y = x \wedge y \neq \bot$, hence y and z are compatible and $(p, q) \in \text{io-act}(y)$. This shows that $\text{io-act}(x) \subseteq \text{io-act}(y)$ and that x is terminal.

Any pure input is terminal. Consider now a pure input i and a compatible input j. Let $\text{io-act}(x) \in \text{i-act}(i)$. It holds that x is compatible with i, hence $i \wedge x \neq \bot$. Since i is pure, $i \wedge j = i$, thus $i \wedge j \wedge x \neq \bot$, and x is also compatible with j, implying that $\text{io-act}(x) \in \text{i-act}(j)$. This shows that $\text{i-act}(i) \subseteq \text{i-act}(j)$ and that i is terminal.

Any terminal IO and input is valid. We prove the case for inputs, the IO case being similar. Let i be a terminal input and j be a compatible pure input (at least one exists), then $\text{i-act}(i) \subseteq \text{i-act}(j)$. Since j is pure, it is also terminal, hence $\text{i-act}(j) \subseteq \text{i-act}(i)$. Hence $\text{i-act}(i) = \text{i-act}(j)$ and i is valid. □

We present a simple algorithm for computing a sufficient set of terminal IOs. This is done by iteratively refining a set P of terminal IOs, starting by assuming that $\{\top\}$ is such a set and using any counterexample to split the IOs:

Algorithm 2 Computing a sufficient set of terminal IOs

```
Input: A Büchi automaton 𝒜
Output: A sufficient set of terminal IOs
P ← {⊤}
for every label x in the automaton do
    for every element y in P do
        if x ∧ y ≠ ⊥ then
            Delete y from P
            Insert x ∧ y in P
            if ¬x ∧ y ≠ ⊥ then insert ¬x ∧ y in P
return P
```

We provide 3 implementations of input selection:
- No precomputation, i.e., return pure inputs/IOs;
- Applying Algorithm 2 twice: for IOs and inputs;
- Use a pure BDD approach to do the previous algorithm; this relies on extra variables to have the loop "**for** *every element* y *in* P" iterate *only* over elements y that satisfy $x \wedge y \neq \bot$.

4.5 Precomputing actions (line 6)

Since computing CPre_i for an input i requires to go through i-act(i), possibly going back to the automaton and iterating through all transitions, it may be beneficial to precompute this set. We provide this step as an optional optimization that is intertwined with the computation of a sufficient set of IOs; for instance, rather than iterating through labels in Algorithm 2, one could iterate through all transitions, and store the set of transitions that are compatible with each terminal IO on the fly.

4.6 Main loop: Picking input-actions (line 8)

We provide several implementations of the input-action picker:
- Return each input-action in turn, until no change has occurred to S while going through all possible input-actions;
- Search for an input-action that is certain to change S. This is based on the concept of *critical input* as presented in [4]. This is reliant on how input-actions are ordered themselves, so we provide multiple options (using a priority queue to prefer inputs that were recently returned, randomize part of the array of input-actions, and randomize the whole array).

4.7 When are we done?

The main algorithm answers either "yes, the formula is realizable" or "don't know." Indeed, for the value of k to provide an exact value, it has to be very large

and reaching a fixed point in the computation becomes impossible in practice. However, it is not necessary to restart the whole algorithm with larger values of k in order to converge towards the correct answer: one can just increase all the components of all the vectors in S (our main set), and go back to the main loop. There are thus two parameters that can be adjusted: the starting value of k and the increment to S each time the loop is restarted.

5 Checking unrealizability of LTL specifications

As mentioned in the preliminaries, for large values of k the **BackwardRealizability** problem is equivalent to a non-zero sum game whose payoff set is the complement of the language of the given automaton. More precisely, for small values of k, a negative answer for the **BackwardRealizability** problem does not imply that the output player does not win the game. Instead, if one is interested in whether the output player wins, a property known as determinacy [23] can be leveraged to instead ask whether a complementary property holds: does the input player win the game?

We thus need to build an automaton $\mathcal{B}$ for which a positive answer to the **BackwardRealizability** translates to the previous property. To do so, we can consider the negation of the input formula, $\neg\phi$, and inverse the roles of the players, that is, swap the inputs and outputs. However, to make sure the semantics of the game is preserved, we also need to have the input player play first, and the output player *react* to the input player's move. To do so, we simply need to have the outputs moved *one step forward* (in the future, in the LTL sense). This can be done directly on the input formula, by putting an X (neXt) operator on each output. This can however make the formula much more complex.

We propose an alternative to this: Obtain the automaton for $\neg\phi$, then push the outputs one state forward. This means that a transition $(p, \langle i, o\rangle, q)$ is translated to a transition (p, i, q), and the output o should be fired from q. In practice, we would need to remember that output, and this would require the construction to consider every state (q, o), augmenting the number of states tremendously. Algorithm 3 for this task, however, tries to minimize the number of states (q, o) necessary by considering nonpure outputs that maximally correspond to a pure input compatible with the original transition label.

Algorithm 3 Modifying $\mathcal{A}$ so that the outputs are shifted forward

Input: A Büchi automaton $\mathcal{A}$ with initial state q_0 and transition set δ
Output: The states S and transitions Δ of the Büchi automaton $\mathcal{B}$
$S, V \leftarrow \{(q_0, \top)\}$
$\Delta \leftarrow \{\}$

```
while V is nonempty do
    Pop (p, o) from V
    for every (p, x, q) ∈ δ do
        y ← x
        while y ≠ ⊥ do    // Iterating through x's minterms focusing on inputs
            Let i be a pure input compatible with y
            o' ← ∃I. x ∧ i    // Extract nonpure output compatible with i
            Add (⟨p, o⟩, o ∧ i, ⟨q, o'⟩) to Δ
            If (q, o') is not in S, add it to S and V
            y ← y ∧ ¬i
return S, Δ
```

6 Benchmarks

6.1 Protocol

For the past few years, the yardstick of performance for synthesis tools is the SYNTCOMP competition [19]. The organizers provide a bank of nearly a thousand LTL formulas, and candidate tools are run with a time limit of one hour on each of them. The tool that solves the most instances in this timeframe wins the competition.

To benchmark our tool, we relied on the 930 LTL formulas that were used in the 2021 SYNTCOMP competition, of which about 60% are realizable. Notably, 864 of all the tests were solved in less than 20 seconds by some tool during the competition, and among the 66 tests left out, 50 were not solved by any tool. This showcases a usual trend of synthesis tools: either they solve an instance fast, or they are unlikely to solve it at all. To better focus on the fine performance differences between the tools, we set a timeout of 60 seconds for all tests.

We compared Acacia-Bonsai against itself using different choices of options, and against Acacia+ [5], Strix [24], and ltlsynt [9,25]. The benchmarks were completed on a Linux computer with the following specifications:

- CPU: Intel® Core™ i7-8700 CPU @ 3.20GHz. This CPU has 6 hyper-threaded cores, meaning that 12 threads can run concurrently. It supports Intel® AVX2, meaning that it has SIMD registers of up to 256 bits.
- Memory: The CPU has 12 MiB of cache, the computer has 16 GiB of DDR4-2666 RAM.

We present some of these results in the form of survival plots (also called cactus plots). They indicate how many instances can be solved within a set time, where the time limit is for each instance. As a rule of thumb, the lower the curve, the better. Since the tool tend to solve a lot of instances under one second, we elected to present these graphics with a logarithmic y-axis.

6.2 Results

The options of Acacia-Bonsai. We compared 25 different configurations of Acacia-Bonsai, in order to single out the best combination of options. We elected to

start with some sensible defaults and test each parameter by diverging from the defaults by a single option each time.

- Preprocessing of the automaton (Section 4.1). This has little impact, although a handful of tests saw an important boost. Overall, the performance was slightly worse with automaton preprocessing, owing to the cost of computing the surely loosing states. We elected to deactivate this option in our best configuration, as this allowed four more tests to pass.
- Boolean states (Section 4.2). This step allowed solving about 5% more tests when activated, globally.
- Vectors and downsets (Section 4.3). Despite a wealth of different implementations, only the k-d tree implementation really stands out, in that it solves 5% fewer tests than the rest. The impact on using SIMD vectors and tailoring downset algorithms to leverage SIMD operations appears to be minimal. This is likely caused by two factors: 1. The increasing ability for modern compilers to automatically identify where SIMD instructions can benefit performances; 2. The relative uselessness of pointwise vector operations in the task at hand.
- Precomputing a sufficient set of inputs and IO (Section 4.4). Computing that set using Algorithm 2 turned out to offer the best performance, solving 23 more tests than using the pure inputs/IOs. The pure BDD approach for this step was slightly more costly.
- Picking input-actions (Section 4.6). The approaches performed equivalently, with a slight edge for the choice of critical inputs without randomizing or priority queue.
- Initial value and increments of k (Section 4.7). We compared several combinations, which had little impact on overall performance, with the best one solving 3 more tests than the worst.
- Unrealizability (Section 5). The following figure shows how the formula-based and the automaton-based approaches to unrealizability compare. We only show the unrealizable tests and add the configuration we use in practice: start two threads, one for each option, and stop as soon as one returns.

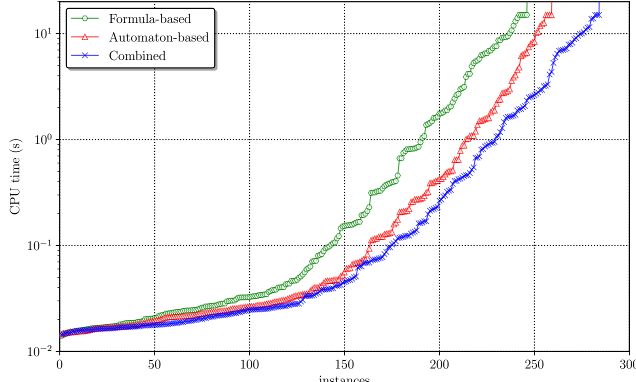

Fig. 2. Reducing unrealizability to realizability. Timeout set at 20 seconds.

Despite the automaton-based approach showing better overall results, we note that this approach provides a larger automaton than the formula-based approach in about 99.5% of the tests. Additionally, the automaton-based approach offers better performances even when looking at the running time *without* the formula-to-automaton part of the process. This seems to indicate that the automaton that is produced is somewhat simpler for the main algorithm.

Acacia-Bonsai and foes. The following plot shows the performance of the tools together. Within our parameters, Acacia-Bonsai solves 699 tests, while Acacia+ solves 560, ltlsynt 703, and Strix 770.

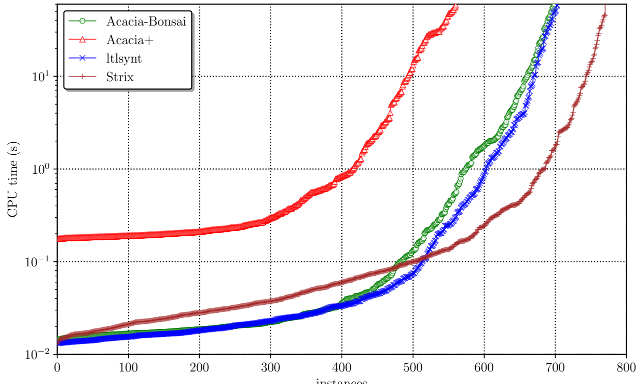

Fig. 3. Survival plot for SYNTCOMP tools and Acacia-Bonsai

Instances solved by one tool but not the other. To better understand the intrinsic algorithmic competitiveness of the different tools, we study which instances were solved by our tool but not the others, and conversely:

- *ltlsynt.* This tool solves 4 more instances than Acacia-Bonsai overall. It solves 61 instances on which Acacia-Bonsai times out, with less than a third of them being unrealizable instances. It would be interesting to implement, within ltlsynt, the unrealizability techniques we describe in Section 5.
- *Strix.* This tool solves 71 more instances than Acacia-Bonsai overall. It solves 124 instances on which Acacia-Bonsai times out, 58% of which are unrealizable. For 90% of these 124 instances, Strix answers in less than 2 seconds. Conversely, of the instances on which Acacia-Bonsai answers while Strix times out, three quarters are solved within two seconds. This naturally hints at the possibility of combining the approaches of the two tools, using parallelization.

7 Conclusion

We provided multiple degrees of freedom in the main algorithm for downset-based LTL realizability and implemented options for each of these degrees. In this paper, we presented the main ideas behind these. Experiments show that this careful reimplementation surpasses the performance of the original Acacia+, making Acacia-Bonsai competitive against modern LTL realizability tools. Along with implementing some optimizations present in previous implementations, we introduced several new ones: reduction of the input-output alphabet, alternative antichain data structures, different strategies for input-picking, and constructing a "shifted automaton" to test unrealizability.

A somewhat disappointing conclusion of our experiments concerns code that makes explicit use of SIMD registers, i.e., large CPU registers that support point-wise vector operations. Our experiments indicate that downset-based algorithms and downset data structures are not able to take full advantage of SIMD. In the future, we plan on investigating data structures for downsets that delay some of their computations in order to better leverage vectorized operations. Such a data structure would not provide better theoretical performances, but would potentially outperform our other data structures.

One surprise that prompts for further investigation is brought by our approach to unrealizability (Section 5): we provided two options for processing the input LTL formula into an automaton that expresses a realizable game iff the original formula was *un*realizable. Although one option consistently produces larger automata than the other, it appears that the downset-based realizability algorithm performs better on the larger automata. A close study of the resulting automata may help in identifying salient features of automata that are easier for the Acacia algorithm.

Lastly, we should note that this reimplementation of Acacia+ is not complete, since a few options of Acacia+ have not yet been included in Acacia-Bonsai yet. One such option consists in decomposing LTL formulas that are conjunctions of subformulas into smaller instances of the realizability problem. We plan on implementing this before the next edition of SYNTCOMP.

Acknowledgements. We would like to thank Véronique Bruyère for recommending the use of k-d trees as a data structure to store and manipulate downsets as well as Clément Tamines for useful conversations on these and alternative data structures. This research was partially funded by the FWO G030020N project "SAILor".

Data-Availability Statement The software presented in this article and the analysed dataset are available as [7]. In addition, the version under study is tagged in the GitHub repository of this software as:

https://github.com/gaperez64/acacia-bonsai/tree/TACAS23

References

1. Ahmed, Z., Benqué, D., Berezin, S., Dahl, A.C.E., Fisher, J., Hall, B.A., Ishtiaq, S., Nanavati, J., Piterman, N., Riechert, M., Skoblov, N.: Bringing LTL model checking to biologists. In: VMCAI. Lecture Notes in Computer Science, vol. 10145, pp. 1–13. Springer (2017)
2. Bansal, S., Li, Y., Tabajara, L.M., Vardi, M.Y.: Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In: AAAI. pp. 9766–9774. AAAI Press (2020)
3. de Berg, M., Cheong, O., van Kreveld, M.J., Overmars, M.H.: Computational geometry: algorithms and applications, 3rd Edition. Springer (2008), https://www.worldcat.org/oclc/227584184
4. Bohy, A.: Antichain based algorithms for the synthesis of reactive systems. Ph.D. thesis, University of Mons (2014)
5. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV. LNCS, vol. 7358, pp. 652–657. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_45
6. Bombara, G., Vasile, C.I., Penedo, F., Yasuoka, H., Belta, C.: A decision tree approach to data classification using signal temporal logic. In: HSCC. pp. 1–10. ACM (2016)
7. Cadilhac, M., Pérez, G.A.: Acacia-Bonsai (TACAS'23 version) (Nov 2022). https://doi.org/10.5281/zenodo.7296659
8. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. In: ICAPS. pp. 621–630. AAAI Press (2019)
9. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω-automata manipulation. In: ATVA. Lecture Notes in Computer Science, vol. 9938, pp. 122–129 (2016)
10. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From LTL and limit-deterministic büchi automata to deterministic parity automata. In: TACAS (1). Lecture Notes in Computer Science, vol. 10205, pp. 426–442 (2017)
11. Esparza, J., Kretínský, J., Sickert, S.: From LTL to deterministic automata - A Safraless compositional approach. Formal Methods Syst. Des. **49**(3), 219–271 (2016). https://doi.org/10.1007/s10703-016-0259-2
12. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of bounded synthesis. In: Legay, A., Margaria, T. (eds.) TACAS. LNCS, vol. 10205, pp. 354–370 (2017). https://doi.org/10.1007/978-3-662-54577-5_20
13. Filiot, E., Jin, N., Raskin, J.: An antichain algorithm for LTL realizability. In: CAV. Lecture Notes in Computer Science, vol. 5643, pp. 263–277. Springer (2009)
14. Geeraerts, G., Goossens, J., Stainer, A.: Synthesising succinct strategies in safety and reachability games. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) RP. LNCS, vol. 8762, pp. 98–111. Springer (2014). https://doi.org/10.1007/978-3-319-11439-2_8
15. Giacomo, G.D., Vardi, M.Y.: $\mathrm{LTL_f}$ and $\mathrm{LDL_f}$ synthesis under partial observability. In: IJCAI. pp. 1044–1050. IJCAI/AAAI Press (2016)
16. Gutierrez, J., Najib, M., Perelli, G., Wooldridge, M.J.: Automated temporal equilibrium analysis: Verification and synthesis of multi-player games. Artif. Intell. **287**, 103353 (2020). https://doi.org/10.1016/j.artint.2020.103353
17. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984. NATO ASI Series, vol. 13, pp. 477–498. Springer (1984). https://doi.org/10.1007/978-3-642-82453-1_17

18. Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J., Sankur, O., Tentrup, L.: The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In: SYNT@CAV. EPTCS, vol. 260, pp. 116–143 (2017)
19. Jacobs, S., Pérez, G.A., Abraham, R., Bruyère, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, K.J., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (SYNTCOMP): 2018-2021. CoRR **abs/2206.00251** (2022). https://doi.org/10.48550/arXiv.2206.00251
20. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: CAV. Lecture Notes in Computer Science, vol. 4144, pp. 31–44. Springer (2006)
21. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining (T). In: ASE. pp. 81–92. IEEE Computer Society (2015)
22. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica **57**(1-2), 3–36 (2020). https://doi.org/10.1007/s00236-019-00349-3
23. Martin, D.A.: Borel determinacy. Annals of Mathematics **102**(2), 363–371 (1975), http://www.jstor.org/stable/1971035
24. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018)
25. Michaud, T., Colange, M.: Reactive synthesis from LTL specification with Spot. In: Proceedings of the 7th Workshop on Synthesis, SYNT@CAV 2018. Electronic Proceedings in Theoretical Computer Science (2018)
26. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3) (2007). https://doi.org/10.2168/LMCS-3(3:5)2007
27. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
28. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989. pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293
29. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 319–327. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21948
30. Shi, Y., Xiao, S., Li, J., Guo, J., Pu, G.: Sat-based automata construction for LTL over finite traces. In: 27th Asia-Pacific Software Engineering Conference, APSEC 2020, Singapore, December 1-4, 2020. pp. 1–10. IEEE (2020). https://doi.org/10.1109/APSEC51365.2020.00008
31. Tomita, T., Ueno, A., Shimakawa, M., Hagihara, S., Yonezaki, N.: Safraless LTL synthesis considering maximal realizability. Acta Informatica **54**(7), 655–692 (2017). https://doi.org/10.1007/s00236-016-0280-3

32. Vardi, M.Y., Wolper, P.: Automata theoretic techniques for modal logics of programs (extended abstract). In: DeMillo, R.A. (ed.) Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA. pp. 446–456. ACM (1984). https://doi.org/10.1145/800057.808711
33. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: A symbolic approach to safety LTL synthesis. In: Strichman, O., Tzoref-Brill, R. (eds.) HVC. LNCS, vol. 10629, pp. 147–162. Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_10

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.