Bounded Synthesis of Reactive Programs*

Carsten Gerstacker, Felix Klein, and Bernd Finkbeiner

Reactive Systems Group, Saarland University, Germany
{gerstacker,fklein,finkbeiner}@cs.uni-saarland.de

Abstract. Most algorithms for the synthesis of reactive systems focus on the construction of finite-state machines rather than actual programs. This often leads to badly structured, unreadable code. In this paper, we present a bounded synthesis approach that automatically constructs, from a given specification in linear-time temporal logic (LTL), a program in Madhusudan's simple imperative language for reactive programs. We develop and compare two principal approaches for the reduction of the synthesis problem to a Boolean constraint satisfaction problem. The first reduction is based on a generalization of bounded synthesis to two-way alternating automata, the second reduction is based on a direct encoding of the program syntax in the constraint system. We report on preliminary experience with a prototype implementation, which indicates that the direct encoding outperforms the automata approach.

1 Introduction

In reactive synthesis, we automatically construct a reactive system, such as the controller of a cyberphysical system, that is guaranteed to satisfy a given specification. The study of the synthesis problem, known also as Church's problem [1], dates back to the 1950s and has, especially in recent years, attracted a lot of attention from both theory and practice. There is a growing number of both tools (cf. [2,3,4,5]) and success stories, such as the synthesis of an arbiter for the AMBA AHB bus, an open industrial standard for the on-chip communication and management of functional blocks in system-on-a-chip (SoC) designs [6].

The practical use of the synthesis tools has, however, so far been limited. A serious criticism is that, compared to code produced by a human programmer, the code produced by the currently available synthesis tools is usually badly structured and, quite simply, unreadable. The reason is that the synthesis tools do not actually synthesize *programs*, but rather much simpler computational models, such as *finite state machines*. As a result, the synthesized code lacks control structures, such as *while* loops, and symbolic operations on program *variables*: everything is flattened out into a huge state graph.

A significant step towards better implementations has been the *bounded synthesis* [7] approach, where the number of states of the synthesized implementation is bounded by a constant. This can be used to construct finite state machines

* Supported by the European Research Council (ERC) Grant OSARES (No. 683300) and by the Saarbrücken Graduate School of Computer Science.

with a *minimal* number of states. Bounded synthesis has also been extended with other structural measures, such as the number of cycles [8]. Bounded synthesis reduces the synthesis problem to a constraint satisfaction problem: the existence of an implementation of bounded size is expressed as a set of Boolean constraints, which can subsequently be solved by a SAT or QBF solver [9]. Bounded synthesis has proven highly effective in finding finite state machines with a *simple* structure. However, existing methods based on bounded synthesis do not make use of syntactical program constructs like loops or variables. The situation is different in the synthesis of sequential programs, where programs have long been studied as the target of synthesis algorithms [10,11,12,13,14]. In particular, in *syntax-guided synthesis* [10], the output of the synthesis algorithm is constrained to programs whose syntax conforms to a given grammar. A first theoretical step in this direction for reactive systems was proposed by Madhusudan [15]. Madhusudan defines a small imperative programming language and shows that the existence of a program in this language with a fixed set of Boolean variables is decidable. For this purpose, the specification is translated into an alternating two-way tree automaton that reads in the syntax tree of a program, simulates its behavior, and accepts all programs whose behavior satisfies the specification. Because the set of variables is fixed in advance, the approach can be used to synthesize programs with a minimal number of variables. However, unlike bounded synthesis, this does not lead to programs that are minimial in other ways, such as the number of states or cycles.

In this paper, we present the first bounded synthesis approach for reactive programs. As in standard bounded synthesis [7], we reduce the synthesis problem to a constraint satisfaction problem. The challenge is to find a constraint system that encodes the existence of a program that satisfies the specification, and that, at the same time, can be solved efficiently. We develop and compare two principal methods. The first method is inspired by Madhusudan's construction in that we also build a two-way tree automaton that recognizes the correct programs. The key difficulty here is that the standard bounded synthesis approach does not work with two-way automata, let alone the alternating two-way automata produced in Madhusudan's construction. We first give a new automata construction that produces universal, instead of alternating, two-way automata. We then generalize bounded synthesis to work on arbitrary graphs, including the run graphs of two-way automata. The second method follows the original bounded synthesis approach more closely. Rather than simulating the execution of the program in the automaton, we encode the existence of both the program and its run graph in the constraint system. The correctness of the synthesized program is ensured, as in the original approach, with a universal (one-way) automaton derived from the specification. Both methods allow us to compute programs that satisfy the given specification and that are minimal in measures such as the size of the program. The two approaches compute the exact same reactive programs, but differ, conceptually, in how much work is done via an automata-theoretic construction vs. in the constraint solving. In the first approach, the verification of the synthesized program is done by the automaton, in the second approach by the constraint solving. Which approach is better? While no method has a clear theoretical advantage over the other, our experiments with a prototype implementation indicate a strong advantage for the second approach.

2 Preliminaries

We denote the Boolean values $\{0,1\}$ by $\mathbb{B}$. The set of non-negative integers is denoted by $\mathbb{N}$ and for $a \in \mathbb{N}$ the set $\{0,1,\ldots,a\}$ is denoted by $[a]$. An *alphabet* Σ is a non-empty finite set of symbols. The elements of an alphabet are called letters. A *infinite word* α over an alphabet Σ is a infinite concatenation $\alpha = \alpha_0\alpha_1\ldots$ of letters of Σ. The set of infinite words is denoted by Σ^ω. With $\alpha_n \in \Sigma$ we access the n-th letter of the word. For an infinite word $\alpha \in \Sigma^\omega$ we define with $\mathrm{Inf}(\alpha)$ the set of states that appear infinitely often in α. A subset of Σ^ω is a *language over infinite words*.

2.1 Implementations

Implementations are arbitrary input-deterministic reactive systems. We fix the finite input and output alphabet $\mathcal{I}$ and $\mathcal{O}$, respectively. A *Mealy machine* is a tuple $\mathcal{M} = (\mathcal{I}, \mathcal{O}, M, m_0, \tau, o)$ where $\mathcal{I}$ is an input-alphabet, $\mathcal{O}$ is an output-alphabet, M is a finite set of states, $m_0 \in M$ is an initial state, $\tau : M \times 2^{\mathcal{I}} \to M$ is a transition function and $o : M \times 2^{\mathcal{I}} \to 2^{\mathcal{O}}$ is an output function. A *system path* over an infinite input sequence $\alpha^{\mathcal{I}}$ is the sequence $m_0m_1\ldots \in M^\omega$ such that $\forall i \in \mathbb{N} : \tau(m_i, \alpha_i^{\mathcal{I}}) = m_{i+1}$. The thereby produced infinite output sequence is defined as $\alpha^{\mathcal{O}} = \alpha_0^{\mathcal{O}}\alpha_1^{\mathcal{O}}\ldots \in (2^{\mathcal{O}})^\omega$, where every element has to match the output function, i.e., $\forall i \in \mathbb{N} : \alpha_i^{\mathcal{O}} = o(m_i, \alpha_i^{\mathcal{I}})$. We say a Mealy machine $\mathcal{M}$ produces a word $\alpha = (\alpha_0^{\mathcal{I}} \cup \alpha_0^{\mathcal{O}})(\alpha_1^{\mathcal{I}} \cup \alpha_1^{\mathcal{O}})\ldots \in (2^{\mathcal{I}\cup\mathcal{O}})^\omega$, iff the output $\alpha^{\mathcal{O}}$ is produced for input $\alpha^{\mathcal{I}}$. We refer to the set of all producible words as the language of $\mathcal{M}$, denoted by $\mathcal{L}(\mathcal{M}) \subseteq (2^{\mathcal{I}\cup\mathcal{O}})^\omega$.

A more succinct representation of implementations are *programs*. The programs we are working with are imperative reactive programs over a fixed set of Boolean variables B and fixed input/output aritys $N_{\mathcal{I}}/N_{\mathcal{O}}$. Our approach builds upon [15] and we use the same syntax and semantics. Let $b \in B$ be a variable and both $\vec{b_{\mathcal{I}}}$ and $\vec{b_{\mathcal{O}}}$ be vectors over multiple variables of size $N_{\mathcal{I}}$ and $N_{\mathcal{O}}$, respectively. The syntax is defined with the following grammar

$$\begin{aligned}\langle \mathit{stmt} \rangle &::= \langle \mathit{stmt} \rangle \,;\, \langle \mathit{stmt} \rangle \mid \mathbf{skip} \mid \mathrm{b} := \langle \mathit{expr} \rangle \mid \mathbf{input}\ \vec{b_{\mathcal{I}}} \mid \mathbf{output}\ \vec{b_{\mathcal{O}}} \\ &\quad\mid\ \mathbf{if}(\langle \mathit{expr} \rangle)\ \mathbf{then}\ \{\langle \mathit{stmt} \rangle\}\ \mathbf{else}\ \{\langle \mathit{stmt} \rangle\} \mid \mathbf{while}(\langle \mathit{expr} \rangle)\{\langle \mathit{stmt} \rangle\} \\ \langle \mathit{expr} \rangle &::= \mathrm{b} \mid \mathbf{tt} \mid \mathbf{ff} \mid (\langle \mathit{expr} \rangle \vee \langle \mathit{expr} \rangle) \mid (\neg\, \langle \mathit{expr} \rangle)\end{aligned}$$

The semantics are the natural one. Our programs start with an initial variable valuation we define to be 0 for all variables. The program then interacts with the environment by the means of input and output statements, i.e., for a vector over Boolean variables $\vec{b}$ the statement "**input** $\vec{b}$" takes an input in $\{0,1\}^{N_{\mathcal{I}}}$ from the environment and updates the values of $\vec{b}$. The statement "**output** $\vec{b}$" outputs the values stored in $\vec{b}$, that is an output in $\{0,1\}^{N_{\mathcal{O}}}$. Therefor a program with

```
while(tt) {
  input (r1, r2);
  if(r1) then {
    r2 = ff
  } else {
    skip
  };
  output (r1, r2)
}
```

Fig. 1. Example-Code

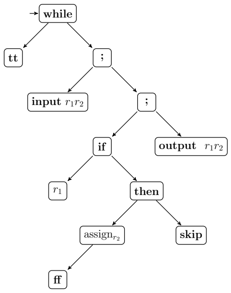


Fig. 2. Example-Program-Tree

input/output arity $N_{\mathcal{I}}/N_{\mathcal{O}}$ requires at least $max(N_{\mathcal{I}}, N_{\mathcal{O}})$ many variables, i.e., $|B| \geq max(N_{\mathcal{I}}, N_{\mathcal{O}})$. Between two input and output statements the program can internally do any number of steps and manipulate the variables using assignments, conditionals and loops. Note that programs are input-deterministic, i.e., a program maps an infinite input sequence $\alpha^{\mathcal{I}} \in (\{0,1\}^{N_{\mathcal{I}}})^{\omega}$ to an infinite output sequence $\alpha^{\mathcal{O}} \in (\{0,1\}^{N_{\mathcal{O}}})^{\omega}$ and we say a program can produce a word $\alpha = (\alpha_0^{\mathcal{I}}\alpha_0^{\mathcal{O}})(\alpha_1^{\mathcal{I}}\alpha_1^{\mathcal{O}})\ldots \in (\{0,1\}^{N_{\mathcal{I}}+N_{\mathcal{O}}})^{\omega}$, iff it maps $\alpha^{\mathcal{I}}$ to $\alpha^{\mathcal{O}}$. We define the language of $\mathcal{T}$, denoted by $\mathcal{L}(\mathcal{T})$, as the set of all producible words. We assume programs to alternate between input and output statements.

We represent our programs as *Σ-labeled binary trees*, i.e., a tuple (T, τ) where $T \subseteq \{L, R\}^*$ is a finite and prefix closed set of nodes and $\tau : T \to \Sigma$ is a labeling function. Based on the defined syntax, we fix the set of labels as

$$\Sigma_P = \{\neg, \vee, ;, \mathbf{if}, \mathbf{then}, \mathbf{while}\} \cup B \cup \{assign_b \mid b \in B\}$$
$$\cup \{\mathbf{input}\ \vec{b} \mid \vec{b} \in B^{N_{\mathcal{I}}}\} \cup \{\mathbf{output}\ \vec{b} \mid \vec{b} \in B^{N_{\mathcal{O}}}\}.$$

We refer to Σ_P-labeled binary trees as *program trees.* If a node has only one subtree we define it to be a the left subtree. Note that our program trees do therefore not contain nodes with only a right subtree. For example, Fig. 1 depicts an arbitrary program and Fig. 2 the corresponding program tree.

We express the current variable valuation as a function $s : B \to \mathbb{B}$. We update variables $\vec{b} \in B^n$ with new values $\vec{v} \in \mathbb{B}^n$ using the following notation:

$$s[\vec{b}/\vec{v}](x) = \begin{cases} v_i & \text{if } b_i = x, \text{for all } i \\ s(x) & \text{otherwise} \end{cases}$$

2.2 Automata

We define *alternating automata over infinite words* as usual, that is a tuple $A = (\Sigma, Q, q_0, \delta, Acc)$ where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \to \mathbb{B}^+(Q)$ is a transition function and $Acc \subseteq Q^{\omega}$ is an acceptance condition.

The *Büchi acceptance condition* BÜCHI(F) on a set of states $F \subseteq Q$ is defined as BÜCHI$(F) = \{q_0q_1 \ldots \in Q^{\omega} \mid \mathrm{Inf}(\alpha) \cap F \neq \emptyset\}$ and F is called the set of accepting states. The *co-Büchi acceptance condition* $\text{COBÜCHI}(F)$ on a set of states $F \subseteq Q$ is defined as $\text{COBÜCHI}(F) = \big\{q_0 q_1 \ldots \in Q^\omega \mid \text{Inf}(\alpha) \cap F = \emptyset\big\}$, where F is called the set of rejecting states. To express combinations of Büchi and co-Büchi expressions we use the *Streett acceptance condition*. Formally, $\text{STREETT}\big(F\big)$ on a set of tuples $F = \{(A_i, G_i)\}_{i\in[k]} \subseteq Q \times Q$ is defined as $\text{STREETT}(F) = \big\{q_0 q_1 \ldots \in Q^\omega \mid \forall i \in [k] : \text{Inf}(\alpha) \cap A_i \neq \emptyset \rightarrow \text{Inf}(\alpha) \cap G_i \neq \emptyset\big\}$. A run with a Streett condition is intuitively accepted, iff for all tuples (A_i, G_i), the set A_i is hit only finitely often or the set G_i is hit infinitely often.

Two-way alternating tree automata are tuple $(\Sigma, P, p_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, Acc)$, where Σ is an input alphabet, P is a finite set of states, $p_0 \in P$ is an initial state, Acc is an acceptance condition, and δ are transition functions of type $\delta_S : P \times \Sigma \times (S \cup \{D\}) \rightarrow \mathbb{B}^+(P \times (S \cup \{U\}))$, for $S \in \{L, R, LR, \emptyset\}$. We introduce $\mu : T \times \{L, R, U\} \rightarrow T \times \{L, R, D\}$ as a function to map states and directions to move in, to the reached states and the matching incoming directions.

$$\begin{array}{ll} \mu(t, L) = (t \cdot L, D) \qquad & \mu(t.L, U) = (t, L) \\ \mu(t, R) = (t \cdot R, D) \qquad & \mu(t.R, U) = (t, R) \end{array}$$

We consider specifications given in linear time-temporal logic (LTL). Such specifications can be translated into non-deterministic Büchi automata or dually into an universal co-Büchi automata as shown in [16]. For an arbitrary specification we denote by A_{spec} and $\overline{A}_{spec}$ the corresponding non-deterministic Büchi and universal co-Büchi automaton, respectively.

3 Automata Construction

We have already argued that programs, as a more succinct representation of implementations, are highly desirable. However, in contrast to Mealy machines, which only dependent on the current state and map an input to a corresponding output, in programs such a direct mapping is not possible. Instead, programs need to be simulated, variables to be altered, expressions to be evaluated and an output statement to be traversed until we produce the corresponding output to the received input. These steps not only depend on the current position in the program but additionally also on the valuation of all variables.

We build upon Madhusudans reactive program synthesis approach [15] were program synthesis is solved by means of two-way alternating Büchi tree automata walking up and down over program trees while keeping track of the current valuation and the state of a given Büchi specification automaton, which is simulated by the input/output produced by traversing the program tree. The automaton accepts a program tree whenever the simulated specification automaton accepts the provided input/output. The constructed automaton, we will further refer to as $\mathcal{A}$, is intersected with two other constructed automata which enforce syntactically correctness and reactivity of the synthesized program, respectively. Then a reactive and syntactically correct program is synthesized by means of an emptiness check of the obtained automaton, involving an exponential blowup to eliminate two-wayness and alternation.

3.1 Two-way Universal Co-Büchi Tree Automaton

We construct a two-way *non-deterministic* Büchi tree automaton $\mathcal{B}$ that is equivalent to $\mathcal{A}$ by using deterministic evaluation of Boolean expressions. We construct $\mathcal{B}$ without an exponential blowup in the state space. We then complement $\mathcal{B}$ into a two-way universal co-Büchi tree automaton convenient for the bounded synthesis approach.

The two-way alternating Büchi tree automaton $\mathcal{A}$ uses universal choices only in relation to Boolean expression evaluation. For example, for **if**, **while** and $assign_b$-statements a Boolean evaluation is needed. In this cases it non-deterministically guesses whether the expression evaluates to 0 or 1 and then universally sends one copy into the Boolean expression, which evaluates to $true$ iff the expression evaluates to the expected value, and one copy to continue the corresponding normal execution. The copy evaluating the Boolean expression walks only downwards and since the subtree corresponding to the Boolean expression is finite, this copy terminates to either $true$ or $false$ after finitely many steps. Instead of using both non-deterministic and universal choices, we evaluate the Boolean subtree deterministically in finitely many steps and then continue the normal execution based on the result of the evaluation.

Note that we not only remove all universal choices but additionally all unnecessary sources of non-determinism. Therefore, besides traversing input- and output-labels, that introduce unavoidable non-determinism, our program simulation is deterministic.

Our automaton $\mathcal{B}$ with the set of states

$$P_{exec} = S \times Q_{spec} \times \mathbb{B}^{N_{\mathcal{I}}} \times \{inp, out\} \times \mathbb{B}$$

$$P^{\mathcal{B}}_{expr} = S \times Q_{spec} \times \mathbb{B}^{N_{\mathcal{I}}} \times \{inp, out\} \times \{\top, \bot\}$$

$$P^{\mathcal{B}} = P^{\mathcal{B}}_{expr} \cup P_{exec}$$

and initial state $p^{\mathcal{B}}_0 = (s_0, q_0, i_0, inp, 0)$, is defined with the transitions shown in Fig. 3, where $s \in S$ is a variable valuation, $q \in Q_{spec}$ the state of the simulated specification automaton, $i \in \mathbb{B}^{N_{\mathcal{I}}}$ the last received input, $m \in \{inp, out\}$ a flag to ensure alternation between inputs and outputs, $r \in \{\top, \bot\}$ the result of a Boolean evaluation and $t \in \{0, 1\}$ a flag for the Büchi condition, which ensures that the specification automaton is simulated for infinite steps and is only set to 1 for a single simulation step after an output statement. We express states corresponding to Boolean evaluations and program execution as $(s, q, i, m, r) \in P^{\mathcal{B}}_{expr}$ and $(s, q, i, m, t) \in P^{\mathcal{B}}_{exec}$, respectively.

The notation reads as follows: If the automaton enters a node with one of the black incoming edges, it can move in the direction of the black outgoing edges, while updating his state corresponding to the annotated update expression, depicted by an enclosing rectangle. Additionally, the automaton needs to fulfill the conditions annotated to the edges it traverses. To express non-determinism we use sets of update expressions, such that each expression represents one possible successor. All state values not contained in the update expression stay the same, except t which is set to 0. When changing from Boolean evaluation to program execution, we copy s, q, i, m and vice versa.

The set of accepting states is defined as

$$F^{\mathcal{B}} = \big\{(s, q, i, m, 1) \mid q \in F_{spec}\big\}$$

A formal construction of $\mathcal{B}$ is given in the full version of the paper [17,18]. Note that $\mathcal{B}$ behaves similar to $\mathcal{A}$ during normal execution and that only Boolean evaluation was altered. Therefore, the state spaces of the automata only differ in the states corresponding to Boolean evaluation and especially the sets of accepting states $F^{\mathcal{A}}$ and $F^{\mathcal{B}}$ are equivalent. Therefore, we can prove the equivalence by showing that both automata visit the same sequences of accepting states and thus accept the same program trees.

Theorem 1 ([17,18]). $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$

We now complement the constructed two-way non-deterministic Büchi automaton into a two-way universal co-Büchi automaton. From this point onwards, we refer with $\mathcal{B}$ to the two-way universal co-Büchi automaton.

Since $\mathcal{A}$ accepts precisely the programs that fail the specification and interact infinitely often with the environment, the complement now only accepts programs that do satisfy the specification or interact finitely often with the environment. We fix the remaining misbehavior by enforcing syntactical correctness and reactiveness.

3.2 Guarantee Syntactical Correctness

Due to the fact that $\mathcal{B}$ was designed to correctly simulate programs of our defined syntax and transitions were only defined for syntax-valid statements, $\mathcal{B}$ implicitly rejects programs that are syntactically invalid. But such programs are only then rejected when their syntactically incorrect statements are traversed in the simulation, therefore $\mathcal{B}$ does not check for syntactically correct subtrees that are unreachable. It is now arguable whether the syntax check is necessary in practice. One could expect programs to be syntactically correct in total and this expectation is in general well-argued. On the other hand, we do perform bounded synthesis, i.e., we search for implementations with a bound on the implementation size and then increment this bound until a valid implementation is found. It is easy to see that programs with unreachable parts can be represented by smaller programs with the same behavior simply by removing unreachable statements. Therefore, with an incremental search one first finds the smallest and thus syntactically correct programs.

3.3 Guarantee Reactiveness

It now remains to guarantee reactiveness of the programs accepted by $\mathcal{B}$. For that purpose, we introduce a two-way universal Büchi automaton $\mathcal{B}_{reactive}$, which

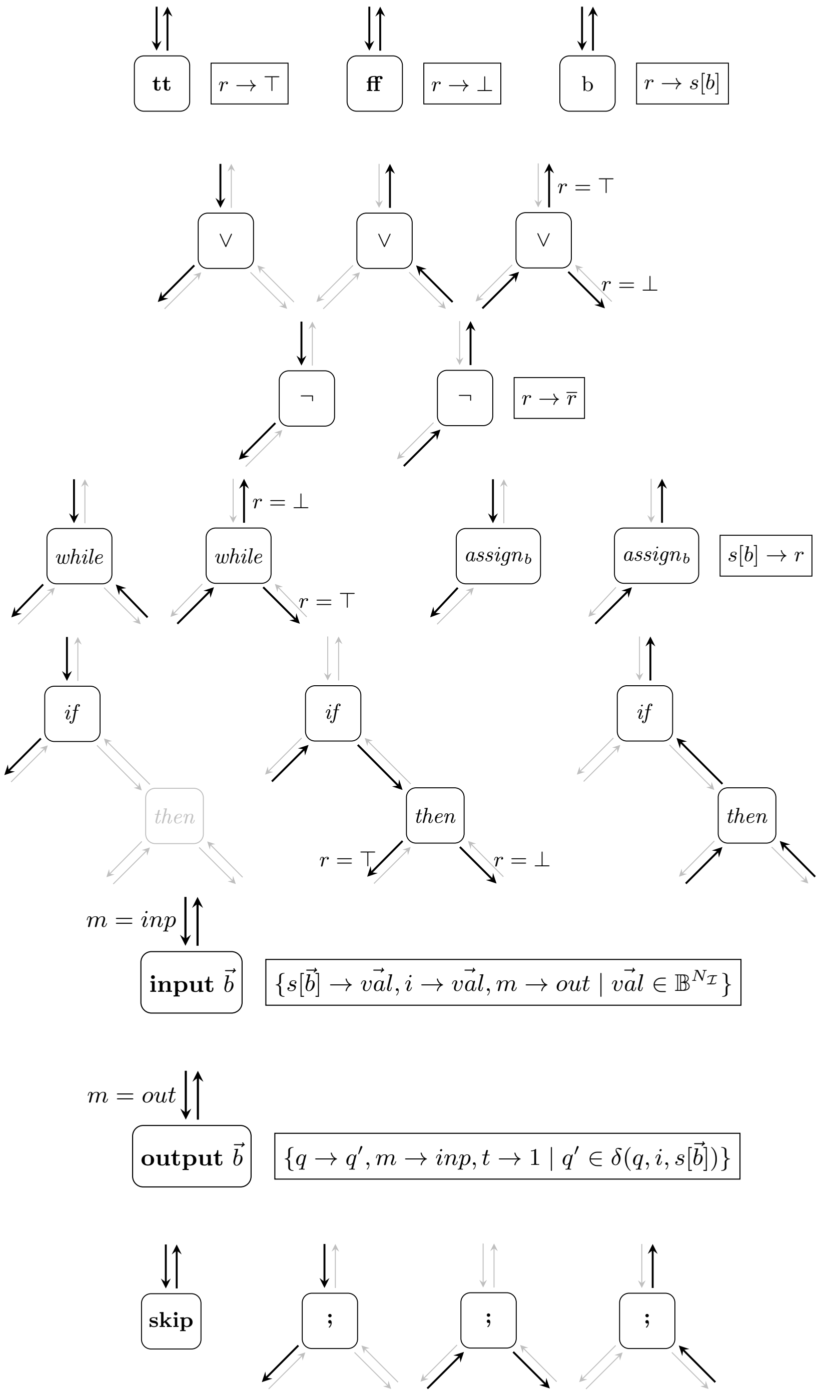


Fig. 3. Semantics of the constructed two-way automata

only accepts program trees that are reactive. This automaton is designed with the exact same states and transitions as $\mathcal{B}$ but with another acceptance condition. The intersection of $\mathcal{B}$ and $\mathcal{B}_{reactive}$ then yields a two-way universal Streett automaton $\mathcal{B}'$. We construct $\mathcal{B}_{reactive}$ with the set of accepting states:

$$F^{\mathcal{B}}_{reactive} = \big\{(s, q, i, m, 1) \mid \forall s, q, i, m\big\}$$

$\mathcal{B}_{reactive}$ accepts a program tree, iff it produces infinitely many outputs on all possible executions. Due to the alternation between input and output statements the program reacts infinitely often with its environment, i.e., it is reactive.

Formally, $\mathcal{B}'$ is the tuple $(\Sigma_P, P^{\mathcal{B}}, \delta^{\mathcal{B}}_L, \delta^{\mathcal{B}}_R, \delta^{\mathcal{B}}_{LR}, \delta^{\mathcal{B}}_{\emptyset}, \mathrm{STREETT}(F^{\mathcal{B}'}))$, where

$$F^{\mathcal{B}'} = \big\{(F^{\mathcal{B}}, \emptyset), (P^{\mathcal{B}}, F^{\mathcal{B}}_{reactive})\big\}.$$

Lemma 1. $\mathcal{L}(\mathcal{B}') = \mathcal{L}(\mathcal{B}) \cap \mathcal{L}(\mathcal{B}_{reactive})$

Proof. Besides the acceptance condition, all three automata are equivalent. The tuples of the Streett condition $(F^{\mathcal{B}}, \emptyset)$ and $(P^{\mathcal{B}}, F^{\mathcal{B}}_{reactive})$ express the co-Büchi and Büchi condition of $\mathcal{B}$ and $\mathcal{B}_{reactive}$, respectively. □

We capture the complete construction by the following theorem.

Theorem 2. *Let B be a finite set of Boolean variables and φ a specification given as LTL-formula. The constructed two-way universal Streett automaton $\mathcal{B}'$ accepts program trees over B that satisfy the specification.*

4 Bounded Synthesis

In this section, we generalize the bounded synthesis approach towards arbitrary universal automata and then apply it to the constructed two-way automaton to synthesize bounded programs.

We fix Q to be a finite set of states. A run graph is a tuple $\mathcal{G} = (V, v_0, E, f)$, where V is a finite set of vertices, v_0 is an initial vertex, $E \subseteq V \times V$ is a set of directed edges and $f : V \to Q$ is a labeling function. A path $\pi = \pi_0\pi_1 \ldots \in V^\omega$ is *contained* in $\mathcal{G}$, denoted by $\pi \in \mathcal{G}$, iff $\forall i \in \mathbb{N} : (\pi_i, \pi_{i+1}) \in E$ and $\pi_0 = v_0$, i.e., a path in the graph starting in the initial vertex. We denote with $f(\pi) = f(\pi_0)f(\pi_1) \ldots \in Q^\omega$ the application of f on every node in the path, i.e., a projection to an infinite sequence of states. We call a vertex v *unreachable*, iff there exists no path $\pi \in \mathcal{G}$ containing v. Let $Acc \subseteq Q^\omega$ be an acceptance condition. We say $\mathcal{G}$ *satisfies* Acc, iff every path of $\mathcal{G}$ satisfies the acceptance condition, i.e., $\forall \pi \in \mathcal{G} : f(\pi) \in Acc$.

Run graphs are used to express all possible runs of a universal automaton on some implementation. This is usually done for universal word automata on Mealy machines, but we need a generalized version to later utilize it for two-way universal tree automata on program trees. Let $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$. We define a run graph $\mathcal{G}^A_{\mathcal{M}} = (V, v_0, E, f)$ of a universal word automaton $A = (\Sigma, Q, q_0, \delta, Acc)$ on a Mealy machine $\mathcal{M} = (\mathcal{I}, \mathcal{O}, M, m_0, \tau, o)$ as an instantiation of the given definition, where

- $V = Q \times M$,
- $v_0 = (q_0, m_0)$,
- $E = \big\{\big((q,m),(q',m')\big) \mid \exists in \in 2^{\mathcal{I}}, out \in 2^{\mathcal{O}} :$ $\tau(m, in) = m' \ \wedge\ o(m, in) = out \ \wedge\ q' \in \delta(q, in \cup out)\big\}$ and
- $f(q,m) = q$.

Since the run graph contains all infinite runs of A on words producible by $\mathcal{M}$, A *accepts* $\mathcal{M}$, iff all runs in $\mathcal{G}^A_{\mathcal{M}}$ are accepting, i.e., $\mathcal{G}^A_{\mathcal{M}}$ satisfies Acc.

For some bound $c \in \mathbb{N}$ we denote $\{0, 1, \ldots, c\}$ by D_c. For a run graph $\mathcal{G} = (V, v_0, E, f)$ and a bound $c \in \mathbb{N}$ a c-bounded annotation function on $\mathcal{G}$ is a function $\lambda : V \to D_c$. An annotation comparison relation of arity n is a family of relations $\triangleright = (\triangleright_0, \triangleright_1, \ldots, \triangleright_{n-1}) \in (2^{Q \times D_c \times D_c})^n$. We refer to $\triangleright_i \subseteq Q \times D_c \times D_c$ as basic comparison relations for $i \in [n]$. We denote the arity with $|\triangleright| = n$. We write $\lambda(v) \triangleright_i \lambda(v')$ for $(f(v), \lambda(v), \lambda(v')) \in \triangleright_i$ and for comparison relations of arity $|\triangleright| = 1$ we omit the index.

We say a path $\pi \in \mathcal{G}$ *satisfies* a comparison relation $\triangleright$ with arity $|\triangleright| = n$, denoted by $\pi \models \triangleright$, iff for every basic comparison relation there exists an annotation function that annotates every node with a value such that the annotated number for all consecutive nodes in the path satisfy the basic comparison relation, i.e., $\forall i \in [n] : \exists \lambda : \forall j \in \mathbb{N} : \lambda(\pi_j) \triangleright_i \lambda(\pi_{j+1})$. For an acceptance condition $Acc \subseteq Q^\omega$ we say a comparison relation $\triangleright$ *expresses* Acc, iff all paths in $\mathcal{G}$ satisfy the relation if and only if the path satisfies the acceptance condition, i.e., $\forall \pi \in \mathcal{G} : \pi \models \triangleright \leftrightarrow f(\pi) \in Acc$. A c-bounded annotation function λ on $\mathcal{G} = (V, v_0, E, f)$ is *valid* for a basic annotation comparison relation $\triangleright \subseteq Q \times D_c \times D_c$, iff for all reachable $v, v' \in V : (v, v') \in E \to \lambda(v) \triangleright \lambda(v')$.

We use the following annotation comparison relations to express Büchi, co-Büchi and Streett acceptance conditions.

- Let $F \subseteq Q$ and $Acc = \text{BÜCHI}(F)$. Then $\triangleright^F_B$ is defined as

$$\lambda(v) \triangleright^F_B \lambda(v') = \begin{cases} true & \text{if } f(v) \in F \\ \lambda(v) > \lambda(v') & \text{if } f(v) \notin F \end{cases}$$

- Let $F \subseteq Q$ and $Acc = \text{CO-BÜCHI}(F)$. Then $\triangleright^F_C$ is defined as

$$\lambda(v) \triangleright^F_C \lambda(v') = \begin{cases} \lambda(v) > \lambda(v') & \text{if } f(v) \in F \\ \lambda(v) \geq \lambda(v') & \text{if } f(v) \notin F \end{cases}$$

- Let $F = \{(A_i, G_i)\}_{i \in [k]} \subseteq 2^{Q \times Q}$ and $Acc = \text{STREETT}(F)$. Then $\triangleright^F_S = (\triangleright^{F,0}_S, \triangleright^{F,1}_S, \ldots, \triangleright^{F,k-1}_S)$ is defined as

$$\lambda(v) \triangleright^{F,i}_S \lambda(v') = \begin{cases} true & \text{if } f(v) \in G_i \\ \lambda(v) > \lambda(v') & \text{if } f(v) \in A_i \wedge f(v) \notin G_i \\ \lambda(v) \geq \lambda(v') & \text{if } f(v) \notin A_i \cup G_i \end{cases}$$

Note that $|\triangleright^F_B| = |\triangleright^F_C| = 1$ and $|\triangleright^F_S| = k$.

Theorem 3 ([7,19]). *Let F be a set, the acceptance condition of A be expressed by $\triangleright_X^F$ with $X \in \{B, C, S\}$, $c \in \mathbb{N}$ a bound and $\mathcal{G}_{\mathcal{M}}^A$ the run graph of A on $\mathcal{M}$.*

If and only if, there exists a valid c-bounded annotation function λ_i on $\mathcal{G}_{\mathcal{M}}^A$ for each basic comparison relation $\triangleright_i$, then $\mathcal{G}_{\mathcal{M}}^A$ satisfies Acc.

4.1 General Bounded Synthesis

In Theorem 3 we saw that the acceptance of a Mealy machine $\mathcal{M}$ by a universal automata A can be expressed by the existence of an annotation comparison relation. To do the same for two-way automata on program trees, we generalize this theorem towards arbitrary run graphs.

Let $\mathcal{A} = (\Sigma_P, P, p_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, Acc)$ be a two-way universal tree automaton and $\mathcal{T} = (T, \tau)$ a program tree. We define the run graph of $\mathcal{A}$ on $\mathcal{T}$ as $\mathcal{G}_{\mathcal{T}}^{\mathcal{A}} = (V, v_0, E, f)$, where

- $V = P \times T \times \{L, R, D\}$,
- $v_0 = (p_0, \epsilon, D)$,
- $E = \big\{\big((p, t, d), (p', t', d')\big) \mid \exists d'' \in \{L, R, U\} : \mu(t, d'') = (t', d')$
  $\wedge (p', d'') \in \delta_t(p, \tau(t), d)\big\}$ and
- $f(p, t, d) = p$.

For the generalized encoding, we use the same construction for the annotation comparison relation as presented in [19] for Street acceptance conditions, which conveniently suffices for the general run graphs. Büchi and co-Büchi then follow as special cases.

Lemma 2 ([17,18]). *For a Streett acceptance condition $Acc = Streett(F)$ with set of tuples of states $F \subseteq 2^{Q \times Q}$ and a run graph $\mathcal{G} = (V, v_0, E, f)$:*

If $\mathcal{G}$ satisfies Acc, then there exists a valid $|V|$-bounded annotation function λ for each basic comparison relation in $\triangleright_S^F$.

Theorem 4. *Let $\mathcal{G} = (V, v_0, E, f)$ be a run graph, $Acc \subseteq Q^\omega$ a Büchi, co-Büchi or Streett acceptance condition expressed by the relation $\triangleright_X$ for $X \in \{B, C, S\}$.*

There exists a valid $|V|$-bounded annotation function λ_i on $\mathcal{G}$ for each basic comparison relation $\triangleright_i$, if and only if $\mathcal{G}$ satisfies Acc.

Proof. " $\Rightarrow$ " : Let $\mathcal{G}$, Acc, $\triangleright$ with arity $|\triangleright| = n$ and c be given and λ_i be a valid c-bounded annotation comparison relation on $\mathcal{G}$ for $\triangleright_i$ for all $i \in [n]$. Let $\pi = \pi_0\pi_1 \ldots \in \mathcal{G}$ be an arbitrary path in $\mathcal{G}$ and $i \in [n]$. Since λ_i is a valid annotation function, $\lambda_i(\pi_0) \triangleright_i \lambda_i(\pi_1) \triangleright_i \ldots$ holds and therefore $\pi \models \triangleright$. Since $\triangleright$ expresses Acc it follows that $f(\pi) \in Acc$, i.e., $\mathcal{G}$ satisfies Acc.
" $\Leftarrow$ " : Lemma 2. □

4.2 General Encoding

We showed that the run graph satisfies an acceptance condition Acc, iff the implementation is accepted by the automaton. We also proved that the satisfaction of Acc by a run graph can be expressed by the existence of valid annotation functions.

We encode these constraints in SAT. The valid implementation can then be extracted from the satisfied encoding. Note that in our definition of program trees the structure was implicitly expressed by the nodes and for the encoding we need to express them explicitly. Therefore, the structure of the tree is encoded with successor functions L and R, expressing the left and right child of a node, respectively. We encode the program tree and the annotation function as uninterpreted functions as explained in the following. We introduce the following variables for arbitrary two-way automata $\mathcal{A}$, program trees $\mathcal{T}$, bounds c and annotation comparison relations $\triangleright$:

- τ_t encodes label l of t with $\log(|\Sigma|)$ many variables, notated as $\tau_t \equiv l$
- L_t iff t has left child (implicitly the next program state $t+1$)
- R_t encodes right the child of $t \in T$ with $\log(|T|)$ many variables
- $\lambda^{\mathbb{B}}_{p,t,d}$ iff state (p,t,d) is reachable in the run graph
- $\lambda^{\#}_{i,p,t,d}$ encodes the i-th annotation of state (p,t,d) with $\log(c)$ many variables. We omit the index i in the encoding

The SAT formula $\Phi^{\mathcal{A},\triangleright}_{\mathcal{T}}$ consists of the following constraints:

- The initial state is reachable and all annotations fulfill the given bound:

$$\lambda^{\mathbb{B}}_{p_0,t_0,D} \wedge \bigwedge_{\substack{p\in P,\\ t\in T,\\ d\in\{L,R,D\}}} \lambda^{\#}_{p,t,d} \leq c$$

- Bounded synthesis encoding

$$\bigwedge_{\substack{p\in P,\\ t\in T,\\ d\in D}} \lambda^{\mathbb{B}}_{p,t,d} \rightarrow \bigwedge_{\sigma\in\Sigma} (\tau_t \equiv \sigma) \rightarrow \bigwedge_{\substack{(p',d'')\in\delta(p,\sigma,d),\\ t'\in T,\\ (\varphi,d')\in\mu'(t,d'',t')}} \varphi \rightarrow \lambda^{\mathbb{B}}_{p',t',d'} \wedge \lambda^{\#}_{p,t,d} \triangleright \lambda^{\#}_{p',t',d'}$$

  $\mu' : T \times D' \times T \rightarrow [\mathbb{B}(L_t, R_t) \times D]$ returns a list of pairs (φ, d'), where the formula φ enforces the tree structure needed to reach p', t', d'.

The encoding checks whether universal properties in the run graph hold. Note that we need to additionally forbid walking up from the root node, which is omitted here.

Theorem 5. *Given a two-way universal tree automaton $\mathcal{A}$ with a Büchi, co-Büchi or Streett acceptance condition Acc expressed by $\triangleright$ and a bound $c \in \mathbb{N}$. The constraint system $\Phi^{\mathcal{A},\triangleright}_{\mathcal{T}}$ is satisfiable, iff there is a program tree $\mathcal{T}$ with size $|\mathcal{T}| \leq \lfloor c/|\mathcal{A}| \rfloor$ that is accepted by $\mathcal{A}$.*

Proof. " $\Rightarrow$ " : Let $\mathcal{T}$ be accepted by $\mathcal{A}$, then with Theorem 4 there exists a valid annotation function λ_i on $\mathcal{G}$ for each $i \in [|\triangleright|]$. Let λ_i be represented by $\lambda^{\#}_i$ and $\lambda^{\mathbb{B}}_i$ be *true* for all reachable states in the run graph $\mathcal{G}$. Then $\Phi^{\mathcal{A},\triangleright}_{\mathcal{T}}$ is satisfied.

" $\Leftarrow$ " : Let $\Phi^{\mathcal{A},\triangleright}_{\mathcal{T}}$ be satisfied. Then there exists a valid annotation function λ_i encoded by $\lambda^{\#}_i$ for each $i \in [|\triangleright|]$ (set $\lambda_i(v) = 0$ for all unreachable states v, i.e., where $\lambda^{\#}_i(v)$ is *false*) that satisfies the encoding. With Theorem 4 the acceptance of $\mathcal{T}$ by $\mathcal{A}$ follows. □

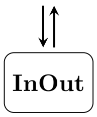
Utilizing this theorem, we now can by means of the encoding $\varPhi_S^{\mathcal{B}'}$ synthesize program trees accepted by $\mathcal{B}'$, i.e., precisely those program trees, which correspond to reactive programs that satisfy the given specification the automaton was constructed with.

Corollary 1. *The SAT encoding $\varPhi_S^{\mathcal{B}'}$ is satisfiable, if and only if there exists a program tree $\mathcal{T}$ with size $|\mathcal{T}| \leq \lfloor c/|\mathcal{B}'| \rfloor$ accepted by $\mathcal{B}'$.*

Size of construction The automaton can be constructed of size $O(2^{|B|+|\varphi|})$, i.e., for a fixed set of Boolean variables the automaton is linear in the size of the specification automaton or exponential in the size of the specification formula. The constructed constraint system $\varPhi_S^{\mathcal{B}'}$ is of size $O(|T| \cdot |\delta| \cdot |\varSigma_P|)$ with x many variables, where $x \in O(|T| \cdot (|T| + |\varSigma_P| + |Q| \cdot log(|Q| \cdot |T|)))$. Note that $|\varSigma_P| \in O(|B|^{N_\mathcal{I}+N_\mathcal{O}})$ grows polynomial in the number of variables for fixed input/output arities.

5 Two-Wayless Encoding

Next, we sketch the second encoding that avoids the detour via universal two-way automata. To this end, we alter the construction in that input- and output-labels collapse to a single *InOut*-label with semantics as follows



$$\boxed{\{\vec{i} \to \vec{val}, q \to q' \mid \vec{val} \in \mathbb{B}^{N_\mathcal{I}}, q' \in \delta(q, \vec{i}, s[\vec{o}])\}}$$

where we use output variables $\vec{o}$ and input variables $\vec{i}$ that correspond to inputs and outputs of the system, respectively. In a nutshell, our new encoding consists of four parts:

1. The first part guesses the program and ensures syntactical correctness.
2. The second part simulates the program for every possible input from every reachable *InOut*-labeled state until it again reaches the next *InOut*-labeled state. Note that every such simulation trace is deterministic once the input, read at the initial *InOut*-labeled state, has been fixed.
3. The third part extracts a simplified transition structure from the resulting execution graph, that consists of direct input labeled transitions from one *InOut*-labeled state to the next one and output labeled states.
4. In the last part, this structure is then verified by a run graph construction that must satisfy the specification, given as universal co-Büchi automaton. To this end, we couple inputs on the edges with the outputs of the successor state to preserve the Mealy semantics of the program.

The first part utilizes a similar structure as used for the previous encoding and thus is skipped for convenience here. To simulate the program in the second part, we introduce the notion of a *valuation* $v \in \mathcal{V}$, where

$$\mathcal{V} = P \times \mathbb{B}^{B \smallsetminus \mathcal{I}} \times \{L, R, U\} \times \mathbb{B}$$

Table 1. Comparison of the general and the two-wayless encoding.

SPECIFICATION	STATES	ADDITIONAL VARIABLES	$\lvert\mathcal{B}'\rvert$	TWO-WAY ENCODING	TWO-WAYLESS ENCODING
$in \leftrightarrow out$	6	0	16	00m16s	00m02s
$in \leftrightarrow \bigcirc\, out$	9	1	64	11m29s	08m34s
latch	10	0	64	>120m	08m07s
2-bit arbiter	10	0	128	66m48s	14m18s

captures the current program state, the current values of all non-input variables, the current direction, and the result of the evaluation of the last Boolean expression, respectively. The simulation of the program is then expressed by a finite execution graph, in which, after fixing a inputs $\vec{i} \in 2^{\mathcal{I}}$, every valuation points to a successor valuation. This successor valuation is unique, except for *InOut*-labeled states, whose successor depends on the next input to be read. The deterministic evaluation follows from the rules of Figure 3 and selects a unique successor for every configuration, accordingly.

In part three, this expression graph then is compressed into a simplified transition structure. To this end, we need for every input and *InOut*-labeled starting valuation, the target *InOut*-labeled valuation that is reached as a result of the deterministic evaluation. In other words, we require to find a shortcut from every such valuation to the next one. We use an inductive chain of constraints to determine this shortcut efficiently. Remember that we only know the unique successor of every valuation which only allows to make one step forward at a time. Hence, we can store for every valuation and input a second shortcut successor, using an additional set of variables, constrainted as follows: if the evaluated successor is *InOut*-labeled, then the shortcut successor must be the same as the evaluated one. Otherwise, it is the same as the shortcut successor of the successor valuation, leading to the desired inductive definition. Furthermore, to ensure a proper induction base, we use an additional ranking on the valuations that bounds the number of steps between two *InOut* labeled valuations. This annotation is realized in a similar fashion as in the previously presented encoding.

With these shortcuts at hand, we then can extract the simplified transition structure, which is verified using a standard run graph encoding as used for classical bounded synthesis. Furthermore, we use an over-approximation to bound the size of the struture and use a reachability annotation that allows the solver to reduce the constraints to those parts as required by the selected solution. The size can, however, also be bound using an explicit bound that is set manually.

Using this separation into four independent steps allows to keep the encoding compact in size, and results in the previously promised performance improvements presented in the next section.

Table 2. Synthesized implementations for the two-wayless encoding.

$in \leftrightarrow out$	$in \leftrightarrow \bigcirc out$	latch	2-bit arbiter

`while(tt) {`
`  out = in;`
`  InOut`
`}`
	`while(tt) {`
	`  out = var;`
	`  var = in;`
	`  InOut`
	`}`
		`while (tt) {`
		`  if (upd) {`
		`    out = in`
		`  } else {`
		`    skip`
		`  };`
		`  InOut`
		`}`
			`while (tt) {`
			`  g0 = g1;`
			`  g1 = not g1;`
			`  InOut`
			`}`

6 Experimental Results

Table 1 compares the general encoding of Section 4.1 and the two-wayless encoding of Section 5 on a selection of standard benchmarks. The table contains the of number of states of the program's syntax tree, the number of additional variables, i.e., variables that are not designated to handle inputs and outputs, the size of the two-way universal Streett automaton, created for the general encoding, and the solving times for both encodings. Table 2 shows the results in terms of the synthesized program trees for the two-wayless encoding. The experiments indicate a strong advantage of the second approach.

7 Conclusions

We introduced a generalized approach to bounded synthesis that is applicable whenever all possible runs of a universal automaton on the possibly produced input/output words of an input-deterministic implementation can be expressed by a run graph. The acceptance of an implementation can then be expressed by the existence of valid annotation functions for an annotation comparison relation that expresses the acceptance of the automaton for Büchi, co-Büchi and Streett acceptance conditions. The existence of valid annotation functions for a run graph is encoded as a SAT query that is satisfiable if and only if there exists an implementation satisfying a given bound that is accepted by the automaton.

For LTL specifications, we constructed a two-way universal Streett automaton which accepts reactive programs that satisfy the specification. We then constructed a run graph that represents all possible runs and applied the generalized bounded synthesis approach. Next, we constructed a SAT query that guesses a reactive program of bounded size as well as valid annotation functions that witnesses the correctness of the synthesized program.

Finally, we merged the previous transformations into an extended encoding that simulates the program direclty via the constraint solver. We evaluated both encodings with the clear result that the encoding avoiding the explicit run graph construction for two-way automata wins in the evaluation.

References

1. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Journal of Symbolic Logic **28**(4) (1963) 289–290
2. Jobstmann, B., Galler, S.J., Weiglhofer, M., Bloem, R.: Anzu: A Tool for Property Synthesis. In Damm, W., Hermanns, H., eds.: CAV, Berlin, Germany. Volume 4590 of LNCS., Springer (2007) 258–262
3. Ehlers, R.: Unbeast: Symbolic Bounded Synthesis. In Abdulla, P.A., Leino, K.R.M., eds.: TACAS, Saarbrücken, Germany. Volume 6605 of LNCS., Springer (2011) 272–275
4. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a Tool for LTL Synthesis. In Madhusudan, P., Seshia, S.A., eds.: CAV, Berkeley, CA, USA. Volume 7358 of LNCS., Springer (2012) 652–657
5. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: An Experimentation Framework for Bounded Synthesis. [20] 325–332
6. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Interactive presentation: Automatic hardware synthesis from specifications: a case study. In Lauwereins, R., Madsen, J., eds.: DATE, Nice, France, EDA Consortium, San Jose, CA, USA (2007) 1188–1193
7. Finkbeiner, B., Schewe, S.: Bounded Synthesis. STTT **15**(5-6) (2013) 519–539
8. Finkbeiner, B., Klein, F.: Bounded Cycle Synthesis. In Chaudhuri, S., Farzan, A., eds.: CAV, Toronto, ON, Canada. Volume 9779 of LNCS., Springer (2016) 118–135
9. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of Bounded Synthesis. In Legay, A., Margaria, T., eds.: TACAS, Uppsala, Sweden. Volume 10205 of LNCS. (2017) 354–370
10. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD, Portland, OR, USA, IEEE (2013) 1–8
11. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In Ball, T., Sagiv, M., eds.: POPL, Austin, TX, USA, ACM (2011) 317–330
12. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In Grove, D., Blackburn, S., eds.: PLDI, Portland, OR, USA, ACM (2015) 619–630
13. Solar-Lezama, A.: Program sketching. STTT **15**(5-6) (2013) 475–495
14. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. STTT **15**(5-6) (2013) 413–431
15. Madhusudan, P.: Synthesizing Reactive Programs. In Bezem, M., ed.: CSL, Bergen, Norway. Volume 12 of LIPIcs., Schloss Dagstuhl (2011) 428–442
16. Vardi, M.Y., Wolper, P.: Reasoning About Infinite Computations. Inf. Comput. **115**(1) (1994) 1–37
17. Gerstacker, C.: Bounded Synthesis of Reactive Programs (2017) Bachelor's Thesis.
18. Gerstacker, C., Klein, F., Finkbeiner, B.: Bounded synthesis of reactive programs. CoRR **1807.09047** (2018)
19. Khalimov, A., Bloem, R.: Bounded Synthesis for Streett, Rabin, and CTL*. [20] 333–352
20. Majumdar, R., Kuncak, V., eds.: CAV, Heidelberg, Germany, Proceedings, Part II. Volume 10427 of LNCS., Springer (2017)