Saarland University
Reactive Systems Group
Department of Computer Science

Bachelor Thesis

Bounded Synthesis of Reactive Programs

Carsten Gerstacker

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Advisor
Felix Klein, M.Sc.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Sebastian Hack, Ph.D.

November 16, 2017

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ______________________ ______________________

(Datum/Date) (Unterschrift/Signature)

Abstract

Bounded Synthesis is the construction of an implementation that satisfies a given temporal specification and a bound on the size of the implementation. The approach presented by Schewe and Finkbeiner introduces Bounded Synthesis of transition systems such as Mealy machines which improves on classic synthesis in that the reduced search space only contains implementations of the specified size.

However, transitions systems are often derived as synthesis artifacts which are too large and structurally complex to be considered comprehensible and therefore motivate the synthesis of more succinct representations of implementations. Programs that are compact and naturally comprehensible are such a desirable representation. Additionally, systems are usually designed by means of high-level representations such as programs.

In this thesis we revisit the bounded synthesis approach and a solution to program synthesis. We then introduce a solution for the bounded synthesis problem for reactive programs based on the revisited approaches.

Contents

Chapter 1

Introduction

Reactive systems are implementations that infinitely often interact with their environment, which means receiving input and corresponding with some output. The systems we consider are *input-deterministic* and *synchronous*, i.e., for some sequence of inputs they always respond with the same sequence of outputs and every input is answered by an output before the next input is received.

One desires implementations to fulfill some specification that restricts the behavior of the implementation. Such specifications in our setting are given as a temporal logic formula over the inputs and outputs of the system. There are two approaches to guarantee the satisfaction of such a specification, namely *verification* and *synthesis*. The verification approach requires a programmer to first implement the system. The implementation is then verified against the specification, i.e., it is checked whether the implementation satisfies the specification for all possible inputs. When the implementation does not fulfill the specification the programmer tries to fix the implementation and verifies the new implementation. This procedure however needs to be repeated until finally an implementation is successfully verified or in the worst-case scenario is repeated forever for an unsatisfiable specification. The synthesis approach eliminates the need to manually design systems. An implementation is synthesized for a given specification such that the synthesized system by construction satisfies the specification. The problem to construct a satisfying implementation for a given specification was first introduced as the synthesis problem by Alonzo Church [3].

However, since the specification only restricts the behavior and not the structure of an implementation, synthesized implementations are often much larger and structurally more complex than manually written implementations. The bounded synthesis approach introduced in [2] synthesizes implementations for a given specification and a given bound on the size of the implementation. Small implementations are often better comprehensible than larger counterparts and often of simpler structure. Also is the synthesis process itself faster since only implementations of bounded size are considered. This approach is further improved towards simple structured implementations in [1], where an additional bound on the contained cycles of an implementation is introduced.

The mentioned synthesis approaches focus on the synthesis of transition systems such as Mealy or Moore machines. However, for some specifications that require more complex implementations, even the smallest satisfying transition systems are incomprehensible. For example are there specifications, that guarantee some properties for a fixed number of clients that interact with the system, such that the size of a valid transition system grows exponential in the size of participating clients. This motivates for the synthesis of more succinct representations of implementations that in some cases can avoid such exponential growth of the synthesized system. Programs for example are a more succinct representation that are naturally used as a high-level description language. An approach that solves the synthesis problem of reactive programs is introduced in [8], where reactive programs over a fixed set of boolean variables that satisfy a given specification are synthesized.

In this thesis we introduce a solution to the bounded synthesis approach for reactive programs. In Chapter 2 we formally define the synthesized implementations, the used automata, the temporal logic used to represent specifications and the synthesis problem. In Chapter 3 we generalize the bounded synthesis approach presented in [2] to be applicable to arbitrary implementation representations. In Chapter 4 we revisit the reactive program synthesis approach introduced in [8] that we then base our construction of an automaton on, that precisely accepts reactive programs over a fixed set of boolean variables and is suitable for the generalized bounded synthesis approach. The construction of this automaton as well as the application of the introduced bounded synthesis approach is then presented in Chapter 5. The last chapter contains the conclusion and further work.

Chapter 2

Preliminaries

We begin with some basic definitions, then introduce the representations of implementations we synthesize, automata over infinite words as well as finite trees and a temporal logic to represent specifications in. Finally we define the synthesis problem for temporal specifications and the bounded synthesis problem with an additional bound on the implementation size, that we solve in thesis for reactive programs.

We denote with $\mathbb{N}$ the set of non-negative integers and for $a, b \in \mathbb{N}$ the set $\{a, a+1, \dots b\}$ is denoted by $[a, b]$. With $[a]$ we denote $[0, a-1]$. An *alphabet* Σ is a non-empty finite set of symbols. The elements of an alphabet are called letters. A *word* over an alphabet Σ is a concatenation $w = w_0 w_1 \dots w_{n-1}$ of letters of Σ, where n defines the length of the word also denoted by $|w|$. The word of length 0 is denoted with ϵ. Σ^* and Σ^ω denote the set of finite and infinite words, respectively. We usually denote finite words with $w \in \Sigma^*$ and infinite words with $\alpha \in \Sigma^\omega$. For an infinite word $\alpha \in \Sigma^\omega$ we access with $\alpha_n \in \Sigma$ the n-th letter of the word, similar for finite words w_n accesses the n-th letter of some finite word $w \in \Sigma^*$. 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^* or Σ^ω is a *language over finite words* or a *language over infinite words*, respectively.

For some set A. A *prefix closed set* $X \subseteq A^*$ is a set that contains for every element also all its prefixes, i.e., $\forall x = x_0 x_1 \dots x_n \in X, i \in [n] : x_0 x_1 \dots x_i \in X$.

2.1 Implementations

Implementations are input-deterministic systems that respond to some input from the environment with some output, i.e., given a sequence of inputs over some input-alphabet they produce a sequence of outputs over some output-alphabet. For this thesis, we fix a finite input-alphabet $\mathcal{I}$ and a finite output-alphabet $\mathcal{O}$. We want to synthesize such systems that additionally fulfill a desired input/output behavior given in the form of a *specification*. We especially focus on *reactive systems*, i.e., systems that infinitely often interact with the environment.

We introduce two kinds of implementations. *Mealy Machines* that are synthesized in the bounded synthesis approach and *programs* over a fixed set of boolean variables that are the target representation of implementations we synthesize in this thesis.

2.1.1 Mealy Machines

Mealy machines are transition systems that respond to input over the input-alphabet $\mathcal{I}$ with output over the output-alphabet $\mathcal{O}$.

Definition 2.1 (Mealy Machines)
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,
- $\tau : M \times 2^{\mathcal{I}} \rightarrow M$ is a transition function and
- $o : M \times 2^{\mathcal{I}} \rightarrow 2^{\mathcal{O}}$ is an output function.

Let $\alpha^{\mathcal{I}} \in (2^{\mathcal{I}})^{\omega}$ be a infinite input sequence. A *system path* over $\alpha^{\mathcal{I}}$ is the sequence $m_0 m_1 \ldots \in M^{\omega}$ 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., $\forall i \in \mathbb{N} : \alpha_i^{\mathcal{O}} = o(m_i, \alpha_i^{\mathcal{I}})$. Note that since the transition

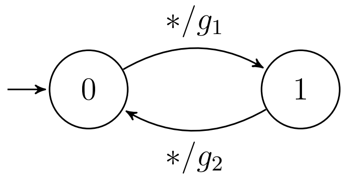

Figure 2.1: Mealy machine of 2-bit arbiter

and output function are deterministic, the system path and the produced output are input-deterministic. We say a Mealy machine $\mathcal{M}$ can produce 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 $\mathcal{L}(\mathcal{M}) \subseteq (2^{\mathcal{I} \cup \mathcal{O}})^\omega$.

As an example we use a 2-bit *arbiter*, i.e., an arbiter for two clients. An arbiter is a system that receives requests from clients and responds with grants to a shared resource, i.e., a system over an input-alphabet $\mathcal{I} = \{r_1, r_2\}$, where r_1 and r_2 are the requests for the first and second client, and an output-alphabet $\mathcal{O} = \{g_1, g_2\}$, where g_1 and g_2 are the grant to access the shared good, respectively. An arbiter adheres to two rules: Every request has to eventually be answered with the corresponding grant and there can never be more than one client accessing the resource at a time. We refer to these rules as the specification for the Mealy machine.

A possible implementation represented as a Mealy machine is shown in Figure 2.1. The two circles represent the states and the labeled arrows between states depict the transition and output function. Each label consists of the input and corresponding output separated by a '/'-symbol, i.e., an arrow between two states $m, m' \in M$ with label 'in/out' represents the transition $\tau(m, in) = m'$ and corresponding output $o(m, in) = out$. A '$*$'-symbol as input means that for every input this transition is valid.
The depicted implementation keeps alternating between its two states and thereby alternates between outputting g_1 and g_2. Every request is granted after maximally two steps and there are never multiple simultaneous grants, therefore the given specification is satisfied. This implementation is easily extended to an arbitrary amount of clients. One simply introduces a new state for each client as well as the corresponding input and output signal. The implementation then is altered such that it cycles through all states giving the grant to a different client in each step. It is easy to see that for this specification of an arbiter the size of the Mealy machine corresponding to a correct implementation grows only linear in the number of clients. So, for this specification a satisfying Mealy machine of acceptable size is easily constructed and also simple structured.

However, this implementation now gives access to every client after a fixed number of steps independently whether the grant was requested or not. An extended version of an arbiter that additionally requires grants to only be given after they have been requested is called a *full arbiter*. A Mealy machine satisfying such a specification is required to remember for each client whether an open request exists or not. Since the encoding of this information requires one state per combination of open requests, the size of the implementation exponentially grows in the number of clients. For a larger amount of clients such a implementation becomes incomprehensible for humans and therefore a more succinct and simpler structured implementation is desirable.

2.1.2 Programs

A more succinct representation of implementations are programs. This naturally arises due to the usage of variables such that even though the *state space* of a program, that contains all possible variable valuations, is exponential in the number of variables the size of the program — measured in lines of code — can be more succinct.

The programs we are working with are imperative programs over a fixed set of boolean variables B and fixed input/output aritys $N_{\mathcal{I}}$, $N_{\mathcal{O}}$. Where $N_{\mathcal{I}}$ and $N_{\mathcal{O}}$ correspond to the size of $\mathcal{I}$ and $\mathcal{O}$, respectively. We again consider *reactive* programs, that infinitely often interact with their environment by receiving input and producing output over said aritys.

Syntax and Semantics

Our programs are defined with the following syntax, where $b \in B$ is a variable and $\vec{b_{\mathcal{I}}}$ and $\vec{b_{\mathcal{O}}}$ are vectors over multiple variables with size $N_{\mathcal{I}}$ and $N_{\mathcal{O}}$ to match the input and output arities, respectively. The syntax is split into statements (*stmt*) and boolean expressions (*expr*).

$$
\begin{array}{lcl}
\langle stmt \rangle & ::= & \langle stmt \rangle \,;\, \langle stmt \rangle \\
 & | & \textbf{skip} \\
 & | & \mathrm{b} := \langle expr \rangle \\
 & | & \textbf{input } \vec{b_{\mathcal{I}}} \\
 & | & \textbf{output } \vec{b_{\mathcal{O}}} \\
 & | & \textbf{if}(\langle expr \rangle) \textbf{ then } \{\langle stmt \rangle\} \textbf{ else } \{\langle stmt \rangle\} \\
 & | & \textbf{while}(\langle expr \rangle)\{\langle stmt \rangle\}
\end{array}
$$

$$
\langle expr \rangle \quad ::= \quad \mathrm{b} \mid \textbf{tt} \mid \textbf{ff} \mid (\langle expr \rangle \lor \langle expr \rangle) \mid (\neg \langle expr \rangle)
$$

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 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 also 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 $\alpha^{\mathcal{O}}$. Also do we assume programs to be synchronous, i.e., to alternate between input and output statements.

Mealy machines and the programs are equally expressive. By that we mean that for every program that can produce a precise set of words there exists a Mealy machine that can also precisely produce this set of words and vice versa. Note that we defined the words produced by Mealy machines to be over the alphabet $2^{\mathcal{I}\cup\mathcal{O}}$ and the words of programs over $\{0,1\}^{N_{\mathcal{I}}+N_{\mathcal{O}}}$. The first alphabet assumes an input or output letter to be set to 1 or 0 when it is an element of $2^{\mathcal{I}\cup\mathcal{O}}$ or not, respectively. The later encodes this explicitly, i.e., every position corresponds to an input or output letter. Therefor both implementations are defined over the same input alphabet $\mathcal{I}$ and output alphabet $\mathcal{O}$. We define the language of $\mathcal{T}$, denoted by $\mathcal{L}(\mathcal{T})$, as the set of all producible words.

For example could the full arbiter represented as a program be implemented by remembering for each client whether a request is still open and cycling through the clients step by step. We therefor introduce the variables $s_0, s_1, \ldots, s_n$ where s_i corresponds to a place for the i-th client, that is set to 1 iff the client has an open request. We also introduce the variables $c_0, c_1, \ldots, c_n$ that represent the current position of a token that is moved to the next place in each step, i.e., c_i is set to 1, iff the token is in the i-th place. The algorithm is then straight forward:

- Read input
- Update the requests $s_0, s_1, \ldots, s_n$
- Output grant g_i iff $c_i \wedge s_i$, i.e., the token is in the i-th place and the i-th client stated a request
- Move the token

Figure 2.2 depicts a 3-bit full arbiter represented as an implementation in the defined syntax. This however can easily be extended to an arbitrary amount of clients. The corresponding n-bit full arbiter is schematically depicted in Figure 2.3.

Since for every new client introduced only a constant number of lines is needed to adapt the program, the implementation remains linear in the number of lines, while the Mealy machine exploded exponentially. The program also remains comprehensible for humans even for large amounts of clients because of its simple structure. Note that in this thesis we do not focus on the structure of synthesized programs but rather refer to the implicit comprehensibility programs provide as a high-level description language. But as further work it might be desirable to consider synthesis of structurally simple programs. For Mealy machines as implementations such a structurally-sensitive approach was already introduced in [1].

Labeled Binary Trees

We represent our programs as Σ_P-labeled binary trees, where the set of labels is

$$\Sigma_P = \{\neg, \wedge, ;, \textbf{if}, \textbf{then}, \textbf{while}\} \cup B \cup \{assign\text{-}b \mid b \in B\}$$
$$\cup \{\textbf{input } \vec{b} \mid \vec{b} \in B^{N_\mathcal{I}}\} \cup \{\textbf{output } \vec{b} \mid \vec{b} \in B^{N_\mathcal{O}}\}.$$

Definition 2.2 (Labeled Binary Trees)
A Σ-labeled binary tree is represented as 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.

```
c$_1$ = 1
while (tt){
    input r$_1$, r$_2$, r$_3$
    s$_1$ = s$_1$$\vee$r$_1$
    s$_2$ = s$_2$$\vee$r$_2$
    s$_3$ = s$_3$$\vee$r$_3$
    output s$_1$$\wedge$c$_1$, s$_2$$\wedge$c$_2$, s$_3$$\wedge$c$_3$
    if c$_1$ then{
        c$_1$ = 0; c$_2$ = 1; s$_1$ = 0;
    }else{
        if c$_2$ then{
            c$_2$ = 0; c$_3$ = 1; s$_2$ = 0;
        }else{
            if c$_3$ then{
                c$_3$ = 0; c$_1$ = 1; s$_3$ = 0;
            }else{
                skip;
            }
        }
    }
}
```

Figure 2.2: 3-bit full arbiter

```
c$_1$ = 1
while (tt):
    input r$_1$, r$_2$, ..., r$_n$
    s$_1$ = s$_1$$\vee$r$_1$
    s$_2$ = s$_2$$\vee$r$_2$
        ...
    s$_n$ = s$_n$$\vee$r$_n$
    output s$_1$$\wedge$c$_1$, s$_2$$\wedge$c$_2$, ... ,s$_n$$\wedge$c$_n$
    if c$_1$:   c$_1$ = 0; c$_2$ = 1; s$_1$ = 0;
    elif c$_2$: c$_2$ = 0; c$_3$ = 1; s$_2$ = 0;
        ...
    elif c$_n$: c$_n$ = 0; c$_1$ = 1; s$_n$ = 0;
```

Figure 2.3: n-bit full arbiter

Labeled binary trees are essentially 2-ary trees but instead of nodes being a subset of $\{1,2\}^*$ we use L and R for the left and right child, respectively. We define the mapping from a program defined with our syntax to a labeled program tree with the following function *tree*. 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.
A term $f(t_1, t_2)$ corresponds to a tree with a f-labeled root where the trees corresponding to t_1 and t_2 are the left and right subtree of the root, respectively. A term $f(t_1)$ corresponds to a tree with a f-labeled root and the tree corresponding to t_1 as left subtree. A term consisting only of a single label f corresponds to a tree with a f-labeled root without any subtrees.

$$\begin{aligned}
tree(b) &= b\\
tree(\mathbf{tt}) &= \mathbf{tt}\\
tree(\mathbf{ff}) &= \mathbf{ff}\\
tree(\varphi_1 \vee \varphi_2) &= \vee(tree(\varphi_1), tree(\varphi_2))\\
tree(\neg\varphi) &= \neg(tree(\varphi))
\end{aligned}$$

$$\begin{aligned}
tree(s; s') &=; (tree(s), tree(s'))\\
tree(\mathbf{skip}) &= \mathbf{skip}\\
tree(\mathbf{input}\ \vec{b}) &= \mathbf{input}\ \vec{b}\\
tree(\mathbf{output}\ \vec{b}) &= \mathbf{output}\ \vec{b}\\
tree(b := e) &= assign\text{-}b(tree(e))\\
tree(\mathbf{if}\ (e)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) &= \mathbf{if}\ (tree(e), \mathbf{then}\ (tree(s_1), tree(s_2)))\\
tree(\mathbf{while}\ (e)\{s\}) &= \mathbf{while}\ (tree(e), tree(s))
\end{aligned}$$

Definition 2.3 (Program trees)
A program tree $\mathcal{T} = (T, \tau)$ is a Σ_P-labeled binary tree.

As an example we depict in Figure 2.4 some arbitrary code we then transform into a program tree. The corresponding program tree is depicted in Figure 2.5.

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

Figure 2.4: Example-Code

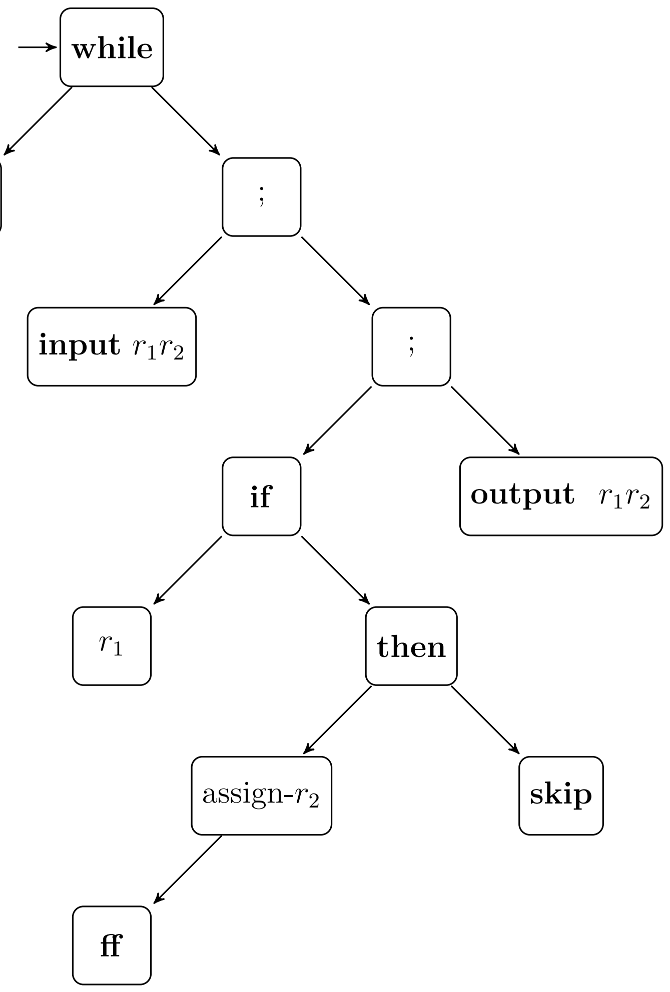

Figure 2.5: Example-Program-Tree

2.2 Automata

In this section we introduce the automata used in this thesis. We first define automata over infinite words as well as their acceptance behavior for both non-deterministic and universal choices. We then use alternating automata to combine both concepts of choices.
We introduce common acceptance conditions of the introduced automata, such as Büchi, co-Büchi and Streett acceptance.
Furthermore we introduce tree automata over finite trees. Usually tree automata like non-deterministic finite tree automata only walk down on a tree. Since our programs are finite labeled binary trees we want to walk over in a reactive manner, that is we want to produce infinite sequences of output based on some infinite input sequence, we need to do infinite steps on an finite tree, that is walking the tree up again. For this purpose we introduce two-way tree automata.

The language of a word/tree automaton A, denoted by $\mathcal{L}(A)$, is the set of all words/trees accepted by that automaton.

2.2.1 Automata on Infinite Words

Definition 2.4 (Automata over Infinite Words)
An automaton over infinite words 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,
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function and
- $Acc \subseteq Q^\omega$ is a accepting condition.

We call A *deterministic*, if $\forall q \in Q, \sigma \in \Sigma : |\delta(q, \sigma)| \leq 1$, that is for every state and input A has at most one successor state. Otherwise we call A *non-deterministic.*

Definition 2.5 (Run on Infinite Words)
For some infinite input word $\alpha \in \Sigma^\omega$ and an automaton $A = (\Sigma, Q, q_0, \delta, Acc)$, we define a run r of A on α as

$$r = r_0 r_1 r_2 \ldots = (q_0, \alpha_0)(q_1, \alpha_1)(q_2, \alpha_2) \ldots \in \Big(Q \times \Sigma\Big)^\omega$$

where $\forall i \in \mathbb{N} : q_{i+1} \in \delta(q_i, \alpha_i)$.

Definition 2.6 (Non-Deterministic Acceptance of Infinite Words)
A non-deterministic automaton A accepts an infinite word α, if

- there is a run r of A on α and
- r is accepting, i.e., $\pi_1(r) \in Acc$.

Where $\pi_1(r)$ is the projection of r to its first tuple elements, i.e., $\pi_1(r) = q_0 q_1 q_2 \ldots \in Q^\omega$.

Non-deterministic automata accept an infinite word iff there exists an accepting run, i.e., in every state of the run there exists a successor that accepts the suffix of the input word. Dually, we introduce *universal* choices, that is that every possible successor needs to accept the suffix of the input word. Universal automata are equally defined but their acceptance is altered.

Definition 2.7 (Universal Acceptance of Infinite Words)
An universal automaton A accepts an infinite word α, if all runs r of A on α are accepting, i.e., $\pi_1(r) \in Acc$.

Alternating Automata

Alternating automata combine the concept of non-deterministic and universal choices. Where non-deterministic choices are understood as guessing some accepting run for an infinite input word, universal choices can be understood as sending a copy of the automaton in each subsequent state and all of those copies then must accept the remainder of the input word. Intuitively, non-deterministic choices can be seen as disjunctions, that accept whenever one of the subsequent states accepts and universal choices as conjunctions where every subsequent state has to accept. This intuition leads to a transition function that maps to not a set of states, but rather to a positive boolean combination over states $\mathbb{B}^+(Q)$, i.e the set defined by the grammar:

$$\phi ::= true \mid false \mid q \mid \phi \vee \phi \mid \phi \wedge \phi$$

We say a set of states $M \subseteq Q$ satisfies such a positive boolean formula iff the boolean combination evaluates to $true$ if all states $q \in M$ are set to $true$ and all states $q' \in Q \setminus M$ not contained in M are set to $false$. We can now define alternating automata.

Definition 2.8 (Alternating Automata over Infinite Words)
An alternating automaton 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.

Notation 2.9 (Membership for Transition Functions of Alternating Automata)
We refer to an alternating automaton as non-deterministic or universal automaton if the positive boolean formulas the transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q)$ maps to consists of only disjunctions or conjunctions, respectively. For such automata we notate by $q' \in \delta(q, \sigma)$ for some states $q, q' \in Q$ and a letter $\sigma \in \Sigma$ that q' is an atom of the conjunction or disjunction $\delta(q, \sigma)$ for universal or non-deterministic automata, respectively.

We define the acceptance of some infinite word by an alternating automaton with the help of a run tree.

Definition 2.10 (Run Tree)
For a word $\alpha \in \Sigma^\omega$ and an alternating automaton $A = (\Sigma, Q, q_0, \delta, Acc)$, a run tree of A on α is an infinite k-ary $(Q \times \Sigma)$-labeled tree represented as a tuple (T, τ) where

- $T \subseteq \{1, \ldots, k\}^* \cup \{1, \ldots, k\}^\omega$ is a infinite and prefix closed set of nodes,
- $\tau : T \to Q \times \Sigma$ is a labeling function and
- let for every node $t \in T$ and $\tau(t) = (q, \alpha_{|t|})$ the set of all successors of t be $S_t = \Big\{t.i \in T \mid i \in \{1, \ldots, k\}\Big\}$. This set needs to satisfy the transition function, that is for the current character $\alpha_{|t|}$ we need $\Big\{q' \mid s \in S_t, \tau(s) = (q', \alpha_{|t|+1})\Big\}$ to satisfy $\delta(q, \alpha_{|t|})$.

Definition 2.11 (Alternating Acceptance of Run Trees)
We say a run tree (T, τ) contains an infinite run $r = r_0 r_1 r_2 \ldots \in (Q \times \Sigma)^\omega$, if there exists a path $t = t_0 t_1 t_2 \ldots \in T$ with $\tau(t_0)\tau(t_0 t_1)\tau(t_0 t_1 t_2) \ldots = r_0 r_1 r_2 \ldots$

A run tree is accepting if all infinite runs it contains are accepting.

An infinite word $\alpha \in \Sigma^\omega$ is accepted by an alternating automaton A, iff there exists an accepting run tree of A on α.

Note that accepting run trees of non-deterministic automata are 1-ary trees, representing *one* accepting run, and run trees of universal automata are unique, representing *all* accepting runs.

Acceptance Conditions

We now define the acceptance conditions that are used in this thesis.

Definition 2.12 (Büchi Acceptance)
The Büchi condition BÜCHI(F) on a set of states $F \subseteq Q$ is defined as

$$\text{BÜCHI}(F) = \Big\{q_0 q_1 \ldots \in Q^\omega \mid \text{Inf}(\alpha) \cap F \neq \emptyset\Big\}.$$

We call F the set of accepting states.

Intuitively, Büchi acceptance accepts a run, iff it hits the set of accepting states infinitely often.

Definition 2.13 (Co-Büchi Acceptance)
The co-Büchi condition COBÜCHI(F) on a set of states $F \subseteq Q$ is defined as

$$\text{COBÜCHI}(F) = \left\{ q_0 q_1 \ldots \in Q^\omega \mid \text{Inf}(\alpha) \cap F = \emptyset \right\}$$

We call F the set of rejecting states.

Intuitively, co-Büchi acceptance accepts a run, iff it hits the set of rejecting states only finitely often.

Büchi and co-Büchi automata are dual, i.e., an alternating Büchi automaton can be complemented by making the acceptance condition co-Büchi and *dualizing* the transition function. Dualizing means to interpret universal choices as non-deterministic ones and vice versa. For example the complementation of an non-deterministic Büchi automaton A yields an universal co-Büchi automaton $\overline{A}$, that accepts all words rejected by A. This conversion becomes intuitive when translated in natural language: "A accepts, iff on some path a set of states is visited infinitely often and $\overline{A}$ accepts, iff on all paths a set of states is visited only finitely often."

Definition 2.14 (Streett Acceptance)
The Streett condition STREETT$\Big(F\Big)$ on a set of tuples $F = \{(A_i, G_i)\}_{i \in [k]} \subseteq Q \times Q$ is defined as

$$\text{STREETT}(F) = \left\{ q_0 q_1 \ldots \in Q^\omega \mid \forall i \in [k] : \text{Inf}(\alpha) \cap A_i \neq \emptyset \to \text{Inf}(\alpha) \cap G_i \neq \emptyset \right\}$$

For Streett acceptance a run is 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.

2.2.2 Automata on Finite Trees

The only trees in this thesis used as input trees for tree automata are finite labeled binary trees, i.e., our program trees defined in Definition 2.3. Therefore we define tree automata only for those labeled binary trees.

Non-Deterministic Finite Tree Automata

We first define non-deterministic tree automata on finite trees. Such an automaton intuitively walks down on a tree by sending a copy of the automaton into every subtree and the states of those copies are chosen non-deterministically.

Definition 2.15 (Non-Deterministic Finite Tree Automata)
A non-deterministic finite tree automata on Σ-labeled trees is a tuple $A = (\Sigma, Q, q_0, \delta_L, \delta_R, \delta_{LR}, F)$ where

- Σ is an input alphabet,
- Q is a finite set of states,
- $q_0 \in Q$ is an initial state,
- $\delta_L, \delta_R : Q \times \Sigma \to 2^Q$ are transition functions for nodes with only a left or right child,
- $\delta_{LR} : Q \times \Sigma \to 2^{Q \times Q}$ is a transition function for nodes with two children and
- $F \subseteq Q$ is a set of accepting states.

Definition 2.16 (Acceptance of Non-Deterministic Tree Automata)
For a non-deterministic tree automaton $A = (\Sigma, Q, q_0, \delta_L, \delta_R, \delta_{LR}, F)$ and a Σ-labeled binary tree $\mathcal{T} = (T, \tau)$ a run of A on $\mathcal{T}$ is a Q-labeled binary tree $\mathcal{T}_r = (T, \tau_r)$, where

- $\tau_r(\epsilon) = q_0$ and
- for all $t \in T$ with label $\tau_r(t) = q$, if
  - t has only a left subtree, then $\tau_r(t.L) \in \delta_L(q, \tau(t))$,
  - t has only a right subtree, then $\tau_r(t.R) \in \delta_R(q, \tau(t))$ and
  - t has both a left and a right subtree, then $(\tau_r(t.L), \tau_r(t.R)) \in \delta_{LR}(q, \tau(t))$.

Note that the structure of the run tree is equal to the input tree and only the labeling function is altered.

A run $\mathcal{T}_r$ is accepted, iff for all leaves $t \in T$ of $\mathcal{T}_r$: $\tau_r(t) \in F$. An input tree $\mathcal{T}$ is accepted by A, iff there exists a accepting run tree of A on $\mathcal{T}$.

Two-Way Alternating Tree Automata

Definition 2.17 (Two-Way Alternating Tree Automata)
A two-way alternating tree automaton is a tuple $(\Sigma, Q, q_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, Acc)$ where

- Σ is an input alphabet,
- Q is a finite set of states,
- $q_0 \in Q$ is an initial state,
- Acc is an acceptance condition,
- δ are transition functions:
  - $\delta_{LR} : Q \times \Sigma \times \{L, R, D\} \to \mathbb{B}^+(Q \times \{L, R, U\})$
  - $\delta_L : Q \times \Sigma \times \{L, D\} \to \mathbb{B}^+(Q \times \{L, U\})$
  - $\delta_R : Q \times \Sigma \times \{R, D\} \to \mathbb{B}^+(Q \times \{R, U\})$
  - $\delta_\emptyset : Q \times \Sigma \times \{D\} \to \mathbb{B}^+(Q \times \{U\})$

Two-way automata allow us to walk up on our input tree, so instead of only walking down on a tree as it is usually done for tree automata it allows us to repeatedly walk over our program tree. Therefor in addition to the current position and the label we currently read in our program tree, the transition function also takes the direction we came from into account and maps to positive boolean formulas over not just states but tuples of states and directions. We also have four transition functions δ_{LR}, δ_L, δ_R and $\delta_\emptyset$ for nodes with a left and right subtree, only a left subtree, only a right subtree – which we won't need since our programs trees do not contain nodes with only a right subtree – and leafs, respectively. Since the applicable transition function depends on the current node in the program tree $t \in T$ we for simplicity denote with δ_t the corresponding transition function, i.e., if t has two subtrees, only a left subtree, only a right subtree or is a leaf, then δ_t refers to δ_{LR}, δ_L, δ_R and $\delta_\emptyset$, respectively.

Definition 2.18 (Run Tree of Two-Way Alternating Tree Automata)
A run of a two-way alternating tree automaton $\mathcal{A} = (\Sigma, Q, q_0, \delta, Acc)$ on a program tree $\mathcal{T} = (T, \tau)$ is an infinite labeled binary tree $\mathcal{T}_R = (T_R, \tau_R)$, where $\tau_R : T_R \to T \times Q \times \{L, R, D\}$ is a labeling function, where labels are tuples that contain the current state of the automaton and program tree as well as the direction it came from. For $\mathcal{T}_R$ it holds that

- $\epsilon \in T_R$, $\tau_R(\epsilon) = (\epsilon, q_0, D)$ and
- $\forall y \in T_R$ with $\tau_R(y) = (t, q, d)$ and $\delta_t(q, \tau(t), d) = \theta :$
  Let $S = \Big\{(q_1, d_1), \ldots, (q_n, d_n)\Big\} \subseteq Q \times \{L, R, U\}$ for $n \leq 2$ be a set that satisfies θ. Then $\forall 1 \leq i \leq n : y \cdot i \in T_R$ and $\tau_R(y \cdot i) = \Big(t', q_i, d'\Big)$ with $(t', d') = \mu(t, d_i)$.

Where μ is a function that maps from a tuple, consisting of a node in our program tree and a direction, to a tuple, containing the node we reach by performing a move in that direction and the direction we came from.

$$\begin{aligned}
&\mu && : T \times \{L, R, U\} \to T \times \{L, R, D\} \\
&\mu(t, L) && = (t \cdot L, D) \\
&\mu(t, R) && = (t \cdot R, D) \\
&\mu(t.L, U) && = (t, L) \\
&\mu(t.R, U) && = (t, R)
\end{aligned}$$

We define run trees as binary trees since the automata we use in this thesis do not have more than two children, but note that in general run trees are of arbitrary arity.

Similar to Definition 2.11 a run tree is accepting, iff all infinite runs it contains are accepting and a tree is accepted when there exists an accepting run tree.

2.3 Linear-Time Temporal Logic

Specifications that our synthesized implementations are to fulfill are usually given in *Linear-time Temporal Logic(LTL)* [9]. LTL is a boolean logic over a set of atomic propositions $\mathcal{I} \cup \mathcal{O}$ extended with temporal operators.

$$\varphi = true \mid a \in \mathcal{I} \cup \mathcal{O} \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathrm{U}\, \varphi$$

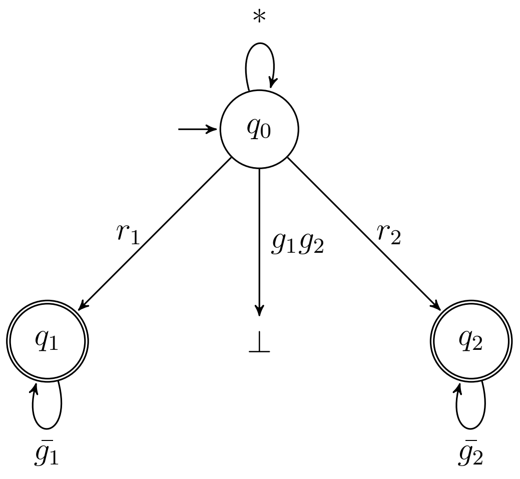

Figure 2.6: Arbiter specification as universal co-Büchi automaton

We denote satisfaction of an LTL-formula φ at a position $n \in \mathbb{N}$ for a word $\alpha \in \Sigma^\omega$ over the alphabet $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$ by $\alpha, n \models \varphi$. Satisfaction is defined by:

- $\alpha, n \models \mathit{true}$
- $\alpha, n \models a$ iff $a \in \alpha_n$
- $\alpha, n \models \neg\varphi$ iff $\alpha, n \not\models \varphi$
- $\alpha, n \models \varphi_1 \vee \varphi_2$ iff $\alpha, n \models \varphi_1$ or $\alpha, n \models \varphi_2$
- $\alpha, n \models \bigcirc\varphi$ iff $\alpha, n+1 \models \varphi$
- $\alpha, n \models \varphi_1 \mathrm{U}\, \varphi_2$ iff $\exists m \geq n : \alpha, m \models \varphi_2$ and $\forall n \leq i < m : \alpha, i \models \varphi_1$

Based on these basic operators we define three additional operators:

- Eventually: $\Diamond\varphi \equiv \mathit{true}\, \mathrm{U}\, \varphi$
- Globally: $\Box\varphi \equiv \neg\Diamond\neg\varphi$
- Weak-Until: $\varphi_1 \mathrm{W}\, \varphi_2 \equiv (\varphi_1 \mathrm{U}\, \varphi_2) \vee \Box\varphi_1$

A word α satisfies an LTL-formula φ, iff $\alpha, 0 \models \varphi$. We define the language of an LTL formula φ to be the set of all infinite words that satisfy φ.

2.3.1 Automata Connection

In this subsection we connect LTL formulas with automata over infinite words and finally with implementations. We first introduce the translation of a formula into a non-deterministic Büchi word automaton.

Theorem 2.19 ([11])
Given an LTL-specification φ. One can construct a non-deterministic Büchi automaton A_φ in time exponential in $|\varphi|$ and of size exponential in $|\varphi|$ that accepts the same language as φ.

By first negating the formula to $\neg\varphi$ and then dualizing the constructed automaton we can also construct an universal co-Büchi word automaton that accepts the same language as φ.

For example, the specification of an 2-bit arbiter given as LTL formula is

$$\varphi = \Box(r_1 \to \Diamond g_1) \wedge \Box(r_2 \to \Diamond g_2) \wedge \Box\neg(g_1 \wedge g_2)$$

where $\Box(r_i \to \Diamond g_i)$ expresses that whenever a request occurs, it has to eventually be granted. The subformula $\Box\neg(g_1 \wedge g_2)$ expresses that two different clients never get granted simultaneously. Figure 2.6 depicts the corresponding universal co-Büchi automaton, where double circled nodes represent rejecting states and transitions to $\perp$ represent instant rejection of a word.

2.4 Synthesis Problem

The synthesis problem for LTL specifications is to synthesize an implementation that fulfills a given specification. A implementation satisfies a specification, iff all words producible by the implementation are accepted by the specification, i.e., the language of the implementation is a subset of the specifications language. In Theorem 2.19 it is stated that such a specification φ can be represented by an universal co-Büchi word automaton A_φ with the same language. Therefore, we can express the satisfaction of a specification φ by an arbitrary Mealy machine $\mathcal{M}$ by the means of the constructed automaton A_φ, i.e., $\mathcal{M}$ satisfies φ, iff $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(A_\varphi)$. Similar for an arbitrary program tree $\mathcal{T}$ it holds $\mathcal{T}$ satisfies φ, iff $\mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(A_\varphi)$.

Since our implementations are input-deterministic and synchronous such that they produce a deterministic output after every input received, the set of possible produced words can be represented as an infinite $2^{\mathcal{I}\cup\mathcal{O}}$-labeled tree, where for every node there exists a child for every possible input and that node is labeled with the received input and the corresponding output of the implementation. Every infinite path in this tree then represents one word producible by the implementation. Note that every implementation corresponds to exactly one tree. One can now lift the universal co-Büchi word automaton A_φ to an universal co-Büchi tree automaton A_φ^T that accepts such trees, iff every word in the tree is accepted by A_φ. The construction of A_φ^T with size exponential in $|\varphi|$ for a given LTL specification φ is shown in [7]. An implementation is then accepted by A_φ^T, iff the produced tree of the implementation is accepted by A_φ^T. For classic synthesis of transition systems such as Mealy machines, where a transition function directly maps an input to an output one yields by means of an emptiness check an accepted tree and then synthesizes a transition system that produces precisely this tree of words.

For programs however this step is more complicated since the internal steps of a program are not represented in the tree and as a result useful information is missing. A program can be synthesized by transforming a synthesized transition system into an equivalent program by simulating the states as variables and encoding the transition function into the program. However, one can also directly synthesize programs as introduced in [8]. We therefor use two-way alternating tree automata that work on program trees and simulate an underlying word automaton representing the specification with the input/output produced while traversing "**input/output**"-statements of the program tree. The construction of such an automaton as well as the synthesis is introduced in Chapter 4.

In this thesis, however, we want to solve the bounded synthesis problem for programs. The bounded synthesis problem is to synthesize an implementation that satisfies a given LTL specification φ and a given bound on the size of the implementation. By means of bounded synthesis one directs the search for satisfying implementations towards small implementations first. An approach for bounded synthesis of transition systems such as Mealy machines was introduced in [2].

In this thesis, we introduce an approach that solves the bounded synthesis problem for reactive programs over a fixed set of boolean variables. We first generalize the bounded synthesis approach introduced for transition systems and then extend the program synthesis approach such that the generalized bounded synthesis approach becomes applicable.

Chapter 3

Bounded Synthesis

3.1 Run Graph

We fix a finite set of states Q.

Definition 3.1 (Run Graph)
A run graph is 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 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., $\forall \pi \in \mathcal{G} : f(\pi) \in Acc$.

A run graph captures all possible runs of an universal automaton with the input/output behavior defined by an arbitrary implementation. We define run graphs for universal

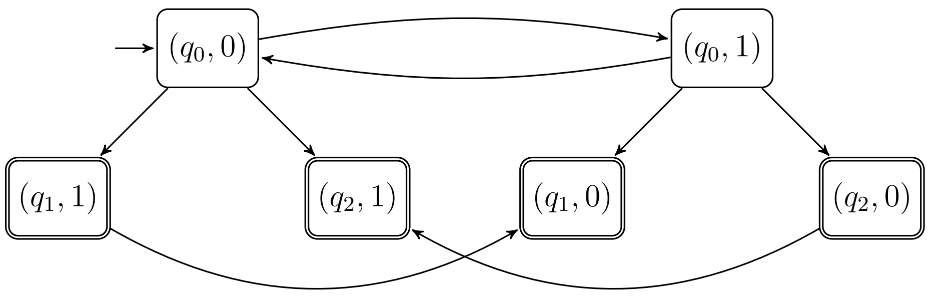

Figure 3.1: Run graph for 2-bit arbiter

word automata on Mealy machines and two-way universal tree automata on program trees.

Definition 3.2 (Run Graph for Mealy Machines)
Let $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$ be a finite alphabet, $A = (\Sigma, Q, q_0, \delta, Acc)$ be a universal word automaton and $\mathcal{M} = (\mathcal{I}, \mathcal{O}, M, m_0, \tau, o)$ a Mealy machine. We define the run graph $\mathcal{G}^A_\mathcal{M} = (V, v_0, E, f)$ of A on $\mathcal{M}$ where

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

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

We have already seen the specification of an 2-bit arbiter given as an universal co-Büchi automaton in Figure 2.6 as well as a possible implementation represented as a Mealy machine in Figure 2.1. The corresponding run graph resulting from this automata and implementation is depicted in Figure 3.1. Since the automaton has a co-Büchi acceptance condition we depict rejecting states by double edges. Note that only the reachable part of the run graph is shown.

The run graph for program trees is again a finite representation of the infinite run tree of $\mathcal{A}$ on $\mathcal{T}$, i.e., every path in $\mathcal{G}^\mathcal{A}_\mathcal{T}$ corresponds to one run in the run tree. Therefor $\mathcal{A}$ accepts $\mathcal{T}$, if and only if $\mathcal{G}^\mathcal{A}_\mathcal{T}$ satisfies *Acc*.

Definition 3.3 (Run Graph for Program Trees)
Let Σ_P be a finite alphabet as defined in Section 2.1.2, $\mathcal{A} = (\Sigma_P, Q, q_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, Acc)$ be a two-way universal tree automaton and $\mathcal{T} = (T, \tau)$ a program tree. We define the run graph $\mathcal{G}_\mathcal{T}^\mathcal{A} = (V, v_0, E, f)$ of $\mathcal{A}$ on $\mathcal{T}$ where

- $V = Q \times T \times \{L, R, D\}$,
- $v_0 = (q_0, \epsilon, D)$,
- $E = \Big\{\big((q,t,d),(q',t',d')\big) \mid \exists d'' \in \{L,R,U\} : \mu(t,d'') = (t',d') \wedge (q',d'') \in \delta_t(q,\tau(t),d)\Big\}$ and
- $f(q,t,d) = q$.

Note that the run graphs of both implementations are unique.

3.2 Annotation Functions and Relations

In this section, we introduce bounded annotation functions, that annotate each state in a run graph with a value, and annotation comparison relations, that are used to express acceptance conditions. We then show that a run graph satisfies an acceptance condition, iff there exist valid bounded annotation functions for a comparison relation that expresses the acceptance condition.

We notate with $D_c = \{0, 1, \ldots, c\} \subset \mathbb{N}$ the finite well-founded set of annotations our annotation functions map to.

Definition 3.4 (Bounded Annotation Function and Comparison Relation)
For a run graph $\mathcal{G} = (V, E, f)$ 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 relation $\triangleright = (\triangleright_0, \triangleright_1, \ldots, \triangleright_{n-1}) \in (2^{Q \times D_c \times D_c})^n$. 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 $\rhd$ with arity $|\rhd| = n$, denoted by $\pi \models \rhd$, iff for every basic comparison relation there exists an annotation function that annotates each node with a value such that the annotated number for all consecutive nodes in the path satisfy the basic comparison relation, i.e.,

$$\forall i \in [n] : \exists \lambda : \forall j \in \mathbb{N} : \lambda(\pi_j) \rhd_i \lambda(\pi_{j+1}).$$

For an acceptance condition $Acc \subseteq Q^\omega$ we say a comparison relation $\rhd$ *expresses Acc*, iff all paths in $\mathcal{G}$ satisfy the relation, if and only if the path satisfies the acceptance condition, i.e.,

$$\forall \pi \in \mathcal{G} : \pi \models \rhd \leftrightarrow f(\pi) \in Acc.$$

Definition 3.5 (Valid Bounded Annotation Function)
A c-bounded annotation function λ on $\mathcal{G} = (V, E, f)$ is valid for a basic annotation comparison relation $\rhd \subseteq Q \times D_c \times D_c$, iff for all reachable $v, v' \in V : (v, v') \in E \rightarrow \lambda(v) \rhd \lambda(v')$.

Theorem 3.6
Let $\mathcal{G} = (V, E, f)$ be a run graph, $Acc \subseteq Q^\omega$ an acceptance condition expressed by an annotation comparison relation $\rhd$ with arity $|\rhd| = n$ and a bound $c \in \mathbb{N}$.

If there exists a valid c-bounded annotation functions λ_i on $\mathcal{G}$ for each basic comparison relation $\rhd_i$, then $\mathcal{G}$ satisfies Acc.

Proof. Let $\mathcal{G}$, Acc, $\rhd$ with arity $|\rhd| = n$ and c be given and λ_i be a valid c-bounded annotation comparison relation on $\mathcal{G}$ for $\rhd_i$ 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, $\lambda_i(\pi_0) \rhd_i \lambda_i(\pi_1) \rhd_i \ldots$ holds and therefore $\pi \models \rhd$. Since $\rhd$ expresses Acc it follows that $f(\pi) \in Acc$, i.e., $\mathcal{G}$ satisfies Acc. □

The other direction does not hold in general as mentioned in [6]. But we can proof the property for individual annotation comparison relations that express some acceptance conditions, namely Büchi, co-Büchi and Streett acceptance conditions. Therefore, we first define comparison relations that express the individual acceptance conditions.

Definition 3.7 (Annotation Comparison Relations)
Let $\mathcal{G} = (V, E, f)$ be run graph.

- Let $F \subseteq Q$ be a set of accepting states and $Acc = \text{BÜCHI}(F)$. The annotation comparison relation $\rhd_B^F$ is defined as

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

  Note $| \rhd_B^F | = 1$.

- Let $F \subseteq Q$ be a set of rejecting states and $Acc = \text{CO-BÜCHI}(F)$. The annotation comparison relation $\rhd_C^F$ is defined as

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

  Note $| \rhd_C^F | = 1$.

- Let $F = \{(A_i, G_i)\}_{i \in [k]} \subseteq 2^{Q \times Q}$ and $Acc = \text{STREETT}(F)$. The annotation comparison relation is defined as $\rhd_S^F = (\rhd_S^{F,0}, \rhd_S^{F,1}, \dots, \rhd_S^{F,k-1})$ where

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

  Note $| \rhd_S^F | = k$.

The comparison relation for Büchi acceptance requires the annotation to count downwards in every step until an accepting state is visited. Therefore, every node can be labeled with the maximal distance to the next accepting state. For co-Büchi acceptance the comparison relation requires the annotation to count downwards when a rejecting state was visited. Therefore, every node can be labeled with the maximal number of rejecting states that can be visited on a path starting in that node. Intuitively, the Streett acceptance comparison relation combines Büchi and co-Büchi comparison relations for each pair $(A, G) \in F$, such that A is considered as rejecting or bad states and G as accepting states. The comparison relation then requires the annotation to count downwards after a rejecting state was visited until an accepting state is visited. Therefore, every node can either be labeled with the maximal number of rejecting states that can be visited on a path starting in that node or whenever there are infinitely many visits to rejecting states, be labeled with the maximal distance until an accepting state is visited. Intuitively, those numbers exists whenever the run graph satisfies the corresponding acceptance condition.

We now proof for every introduced annotation comparison relation that they express their corresponding acceptance condition. Each proof consists of first a lemma to show that the satisfaction of Acc by $\mathcal{G}$ implies the existence of valid bounded annotation functions and finally a theorem to proof that the corresponding acceptance condition is expressed by that relation. Note that for a proof of expression of an acceptance condition by a comparison relation it is sufficient for every basic relation $\triangleright$ to find for every path π, one annotation function λ, such that $\pi \models \triangleright$. However, we proof that there exists a valid annotation function for every basic relation, i.e., one function that holds on every path instead of a function for each path. Whether this quantifier swap is possible in general or not is not further discussed in this thesis, however it is an interesting question left open for further work. We later make use of this property when proofing the inverse direction of Theorem 3.6 for the introduced comparison relations.

Lemma 3.8
For a Büchi acceptance condition $Acc = BÜCHI(F)$ with accepting states $F \subseteq Q$ and a run graph $\mathcal{G} = (V, E, f)$:

If $\mathcal{G}$ satisfies Acc, then there exists a valid $|V|$-bounded annotation function λ for $\triangleright_B^F$.

Proof. Let $F \subseteq Q$ be a set of states, Acc = BÜCHI(F) an acceptance condition and $\mathcal{G} = (V, E, f)$ a run graph satisfying Acc.

Construct the valid c-bounded annotation function λ that maps to each vertex v the maximal distance to a vertex v' with $f(v') = q$ for some $q \in F$, this maximal distance exists since otherwise there would exist an infinite path π without any visits to F and therefore $\pi \notin Acc$ which contradicts the assumption. We define $\lambda(v) = 0$ for all unreachable states $v \in V$. The bound c is the maximal number annotated to a vertex of $\mathcal{G}$. $\square$

Theorem 3.9
For a Büchi acceptance condition $Acc = BÜCHI(F)$ with accepting states $F \subseteq Q$:

$$\triangleright_B^F \text{ expresses Büchi acceptance.}$$

Proof. Let F be a set of states, $Acc = \text{BÜCHI}(F)$ an acceptance condition and $\pi \in \mathcal{G}$ an arbitrary path in a run graph $\mathcal{G} = (V, E, f)$. We prove $\exists \lambda : \forall i \in \mathbb{N} : \lambda(\pi_i) \triangleright_B^F \lambda(\pi_{i+1}) \Leftrightarrow f(\pi) \in Acc$.

$" \Rightarrow "$:
Let λ be a c-bounded annotation function that satisfies $\forall i \in \mathbb{N} : \lambda(\pi_i) \triangleright_B^F \lambda(\pi_{i+1})$. Proof by contradiction. Assume $\pi \notin Acc$. This means there exists a position $n \in \mathbb{N}$ such that $\forall m > n : f(\pi_m) \notin F$. Therefore by definition of $\triangleright_B^F$, it holds $\lambda(\pi_m) < \lambda(\pi_{m+1}) < \ldots$, i.e., there exists an infinite descending chain. This is a contradiction since the domain D_c is well-founded. $\lightning$

$" \Leftarrow "$:
Let $f(\pi) \in Acc$. More precisely since π is an arbitrary path, this property holds for every path in $\mathcal{G}$, i.e., $\mathcal{G}$ satisfies Acc. With Lemma 3.8 there exists a valid c-bounded annotation function λ. $\square$

Lemma 3.10
For a Co-Büchi acceptance condition $Acc = CO\text{-}BÜCHI(F)$ with rejecting states $F \subseteq Q$ and a run graph $\mathcal{G} = (V, E, f)$:

If $\mathcal{G}$ satisfies Acc, then there exists a valid $|V|$-bounded annotation function λ for $\triangleright_C^F$.

Proof. Let $F \subseteq Q$ be a set of states, $Acc = \text{CO-BÜCHI}(F)$ an acceptance condition and $\mathcal{G} = (V, E, f)$ a run graph satisfying Acc.

Construct the c-bounded annotation function λ that maps to each vertex v of $\mathcal{G}$ the maximal amount of visits to rejecting vertices, i.e., a vertex v' with $f(v') = q$ for some $q \in F$, for all paths starting in v. This maximal number of visits exists, since otherwise there would exist a path π in $\mathcal{G}$ with infinite visits to rejecting states and therefore $f(\pi) \notin Acc$ which contradicts the assumptions. We define $\lambda(v) = 0$ for all unreachable states $v \in V$. The bound c is the maximal annotated number to a vertex of $\mathcal{G}$. $\square$

Theorem 3.11

For a Co-Büchi acceptance condition Acc = CO-BÜCHI(F) with rejecting states $F \subseteq Q$:

$$\triangleright_C^F \textit{ expresses co-Büchi acceptance.}$$

Proof. Let F be a set of states, $Acc = \text{CO-BÜCHI}(F)$ an acceptance condition and $\pi \in \mathcal{G}$ an arbitrary path in a run graph $\mathcal{G} = (V, E, f)$. We prove $\exists \lambda : \forall i \in \mathbb{N} : \lambda(\pi_i) \triangleright_B^F \lambda(\pi_{i+1}) \Leftrightarrow f(\pi) \in Acc$.

" $\Rightarrow$ " :
Let λ be a c-bounded annotation function that satisfies $\forall i \in \mathbb{N} : \lambda(\pi_i) \triangleright_C^F \lambda(\pi_{i+1})$. Proof by contradiction. Assume $f(\pi) \notin Acc$. This means there exists a state $q \in F$ that is visited infinitely often, i.e., $\forall n \in \mathbb{N} : \exists m \geq n : f(\pi_m) = q$. Let $I = i_1 i_2 \ldots \in \mathbb{N}^\omega$ with $\forall j \in \mathbb{N} : f(\pi_{i_j}) = q \wedge i_j < i_{j+1}$ be the infinite ordered sequence of all indexes where q occurs in $f(\pi)$. By definition of $\triangleright_C^F$ it holds that $\lambda(\pi_0) \leq \lambda(\pi_1) \leq \ldots \leq \lambda(\pi_{i_1}) < \lambda(\pi_{i_1+1}) \leq \ldots \leq \lambda(\pi_{i_2}) < \lambda(\pi_{i_2+1}) \leq \ldots$, i.e., an infinite descending chain, which is a contradiction since the domain D_c is well-founded. ↯

" $\Leftarrow$ " :
Let $f(\pi) \in Acc$. More precisely since π is an arbitrary path, this property holds for every path in $\mathcal{G}$, i.e., $\mathcal{G}$ satisfies Acc. With Lemma 3.10 there exists a valid c-bounded annotation function λ. □

Lemma 3.12

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

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

Proof. Let $F = \{(A_i, G_i)\}_{i \in [k]}$ be a set of tuples of states, $Acc = \text{STREETT}(F)$ an acceptance condition and $\mathcal{G} = (V, E, f)$ a run graph satisfying Acc.

We construct the k different valid c-bounded annotation functions λ_i. We first define $\lambda_i(v) = 0$ for all unreachable states $v \in V$ and then remove those states from $\mathcal{G}$. We then define each $\lambda_i(v)$ for all $i \in [k]$ with:

- $\forall v \in V$ with $f(v) \in G_i : \lambda_i(v) = 0$.

- Remove outgoing edges for states $v \in V$ with $f(v) \in G_i$. That results in $\mathcal{G}' = (V, E', f)$, where $E' = E \setminus \{(v, v') \in E \mid f(v) \in G_i\}$. $\mathcal{G}'$ contains no *strongly connected components* (SCCs) that have a vertex v with $f(v) \in A_i \cup G_i$. For G_i it is obvious since all outgoing edges are removed. For A_i: Assume there is an SCC containing a state v where $f(v) \in A_i$, then there exists an infinite path π inside this SCC that infinitely often visits A_i without infinite visits to the corresponding set G_i and therefore $f(\pi) \notin Acc$ which contradicts the assumption.

- Now let $\mathcal{S}$ be the set off all SCCs of $\mathcal{G}'$. We define $\mathcal{G}'' = (V'', E'', f)$. Where $V'' = \mathcal{S} \cup \{\{v\} \mid v \notin \bigcup_{S \in \mathcal{S}} S\}$ and $E'' = \{(S_1, S_2) \mid \exists v_1 \in S_1, v_2 \in S_2 : S_1 \neq S_2 \wedge (v_1, v_2) \in E'\}$.
  $\mathcal{G}''$ is a directed acyclic graph and therefor all paths in $\mathcal{G}''$ are of finite length and there only exist finitely many paths.

- We define $nb_i(\pi) : (V'')^* \to \mathbb{N}$ as the max number of bad states visited on some path $\pi = S_1, \ldots, S_m$. Let $Occ(\pi)$ denote the set of occurring SCCs in π. We formally define $nb_i(\pi) = \left|\bigcup_{S \in Occ(\pi)} S \cap \{v \in V \mid f(v) \in A_i\}\right|$.

- $\forall v \in S \in V''$ with $f(v) \notin G_i : \lambda_i(v) = \max(\{nb_i(\pi) \mid \pi \textit{ is a path from } S\})$.

□

Theorem 3.13
For a Streett acceptance condition $Acc = Streett(F)$ with set of tuples of states $F \subseteq 2^{Q \times Q}$:

$$\rhd_S^F \textit{ expresses Streett acceptance.}$$

Proof. Let $F = \{(A_i, G_i)\}_{i \in [k]}$ be a set of tuples of states, $Acc = \text{STREETT}(F)$ an acceptance condition and $\pi \in \mathcal{G}$ an arbitrary path in a run graph $\mathcal{G} = (V, E, f)$. We proof $\forall i \in [k] : \exists \lambda_i : \forall j \in \mathbb{N} : \lambda_i(\pi_j) \rhd_B^{F,i} \lambda_i(\pi_{j+1}) \Leftrightarrow f(\pi) \in Acc$.

$" \Rightarrow "$:
Let λ_i be a c-bounded annotation function that satisfies $\forall j \in \mathbb{N} : \lambda_i(\pi_j) \rhd_S^{F,i} \lambda_i(\pi_{j+1})$ for all $i \in [k]$. Proof by contradiction. Assume $\pi \notin Acc$, i.e., $\exists i \in [k] : \text{Inf}(f(\pi)) \cap A_i \neq \emptyset \wedge \text{Inf}(f(\pi)) \cap G_i = \emptyset$. So there exists a $q \in A_i$ that is visited infinitely often and all $q' \in G_i$ are visited only finitely often. Therefore there exists a $n \in \mathbb{N}$ such that $\forall m \geq n : f(\pi_m) \notin G_i$. Let $I = i_1 i_2 \ldots \in (\mathbb{N} \setminus [n])^\omega$ with $\forall j \in \mathbb{N} : f(\pi_{i_j}) = q \wedge i_j < i_{j+1}$ be the infinite ordered sequence of all indexes bigger that n, where q occurs in $f(\pi)$. By definition of $\rhd_S^{F,i}$ it holds that $\lambda_i(\pi_n) \leq \lambda_i(\pi_{n+1}) \leq \ldots \leq \lambda_i(\pi_{i_1}) < \lambda_i(\pi_{i_1+1}) \leq \ldots \leq$

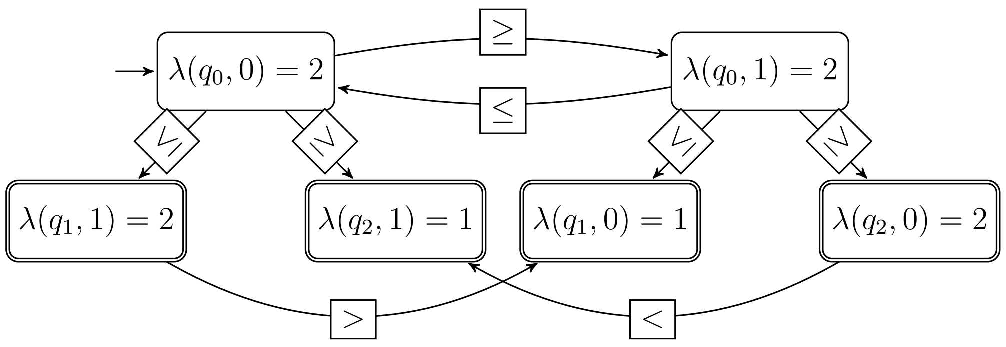

Figure 3.2: Annotated run graph of $\mathcal{A}$ on $\mathcal{T}$

$\lambda_i(\pi_{i_2}) < \lambda_i(\pi_{i_2+1}) \leq \ldots$, i.e., an infinite descending chain, which is a contradiction since the domain D_c is well-founded. $\lightning$

" $\Leftarrow$ " :
Let $\pi \in \text{STREETT}(F)$. More precisely since π is an arbitrary path, this property holds for every path in $\mathcal{G}$, i.e., $\mathcal{G}$ satisfies *Acc*. With Lemma 3.12 there exists a valid c-bounded annotation function λ_i for each basic comparison relation. $\square$

We finally capture the overall results in the following theorem.

Theorem 3.14
Let $\mathcal{G} = (V, E, f)$ 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. " $\Rightarrow$ " : Theorem 3.6.
" $\Leftarrow$ " : Lemma 3.8, Lemma 3.10 and Lemma 3.12. $\square$

Note that this results have already been shown for Mealy automata in [2] and [6]. We solely generalized the results for generalized run graphs, while the proofs correlate to the known results.

Reconsidering our example run graph depicted in Figure 3.1, there needs to exist a bounded annotation function that satisfies the co-Büchi comparison relation. Indeed there exists a 2-bounded annotation function as depicted in Figure 3.2. Note that the relation between two nodes is annotated on each edge and all unreachable nodes are labeled with 0.

3.3 Constraint-Based Bounded Synthesis

We have shown how to construct a run graph for universal word automata on Mealy machines and two-way universal tree automata on program trees. Those run graphs satisfy an acceptance condition *Acc*, iff the corresponding implementation is accepted by the corresponding automaton. We also proved that the satisfaction of *Acc* by a run graph can be expressed by the existence of valid annotation functions for an annotation comparison relation that expresses *Acc*.

We now introduce SAT encodings for both cases, that guess an implementation and simultaneously valid annotation functions, that witness the correctness of the implementation. The SAT encodings are satisfiable, iff there exists a correct implementation that fulfills a given size bound. The valid implementation can then be derived from the satisfied encoding.

3.3.1 Mealy Machines

Based on an universal specification automaton $\mathcal{A} = (\Sigma, Q, q_0, \delta, Acc)$ over the alphabet $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$ where *Acc* is a acceptance condition expressed by $\triangleright$ with arity $|\triangleright| = n$ and a bound $c \in \mathbb{N}$, we construct a constraint system that, when solved by a SAT-solver, yields an implementation represented as a Mealy machine $\mathcal{M} = (\mathcal{I}, \mathcal{O}, M, m_0, \tau)$ with size $|\mathcal{M}| \leq \lfloor c/|\mathcal{A}| \rfloor$ and valid c-bounded annotation functions λ_i for $i \in [n]$ that are witnesses for the implementation. We encode the Mealy machine and the annotation function as uninterpreted functions as explained in the following. We introduce the following variables:

- $\tau(m, \nu, m')$ for all $m, m' \in M$ and $\nu \in 2^{\mathcal{I}}$, iff there is a transition from m with input ν to m'.
- $o(m, \nu, x)$ for all $m \in M, \nu \in 2^{\mathcal{I}}$ and $x \in \mathcal{O}$, iff the transition from m for input ν is labeled with x.
- $\lambda_i^{\mathbb{B}}(q, m)$ for all $i \in [n]$, $m \in M$ and $q \in Q$, iff (q, m) is reachable in the run graph.
- $\lambda_i^{\#}(q, m, x)$ for all $i \in [n], m \in M, q \in Q$ and $0 \leq x < log(c)$, where the annotated number to node (q, m) is encoded as a binary number with $log(c)$ many bits. We

denote with $\lambda_i^{\#}(q,m) \circ k$ for $\circ \in \{<,\leq,=,\geq,>\}$ the relation to some value k or other annotations $\lambda_i^{\#}(q',m')$.

The SAT formula $\Phi_{\mathcal{M}}^{\mathcal{A},\triangleright}$ is build based on the following constraints:

- Transitions of $\mathcal{M}$ are unambiguous:

  $$\bigwedge_{m\in M,\nu\in 2^{\mathcal{I}}} \mathit{exactlyOne}\Big(\{\tau(m,\nu,m') \mid m' \in M\}\Big)$$

  where $\mathit{exactlyOne} : X \to \mathbb{B}(X)$ maps a set to a SAT query to ensure that only one element of the set is true while all remaining are $false$.

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

  $$\bigwedge_{i\in[n]} \lambda_i^{\mathbb{B}}(q_0,m_0) \wedge \bigwedge_{i\in[n],m\in M,q\in Q} \lambda_i^{\#}(q,m) \leq c$$

- All reachable states in the run graph are annotated and fulfill the comparison relation

  $$\bigwedge_{i\in[n],m\in M,q\in Q} \lambda_i^{\mathbb{B}}(q,m) \to \bigwedge_{\sigma\in\Sigma} \mathit{label}(m,\sigma) \to \bigwedge_{m'\in M} \tau(m,\mathcal{I}\cap\sigma,m')$$

  $$\to \bigwedge_{q'\in\delta(q,\sigma)} \lambda_i^{\mathbb{B}}(q',m') \wedge \lambda_i^{\#}(q,m) \triangleright_i \lambda_i^{\#}(q',m')$$

  where $\mathit{label}(m,\sigma) = \bigwedge_{x\in\mathcal{O}\cap\sigma} o(t,\mathcal{I}\cap\sigma,x) \wedge \bigwedge_{x\notin\mathcal{O}\cap\sigma} \neg o(t,\mathcal{I}\cap\sigma,x)$ fixes the label of each transition.

Note this SAT-encoding stems from [1] and is modified to encode multiple annotation functions. The encoding checks whether universal properties in the run graph hold. In [6] they use an SMT encoding to check such universal properties for Büchi and Streett automata. Both encodings are equivalent and we can therefore use the SAT encoding to also check Büchi and Streett conditions. We collect the results for both papers in the following theorem.

Theorem 3.15 ([1] and [6])
Given a universal word 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 $\Phi_{\mathcal{M}}^{\mathcal{A},\triangleright}$ is satisfiable, iff there is a Mealy machine $\mathcal{M}$ with size $|\mathcal{M}| \leq \lfloor c/|\mathcal{A}| \rfloor$ that is accepted by $\mathcal{A}$.

3.3.2 Program Trees

Based on a two-way universal specification automaton $\mathcal{A} = (\Sigma, Q, q_0, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, Acc)$ over an alphabet Σ where Acc is an acceptance condition expressed by $\triangleright$ with arity $|\triangleright| = n$ and a bound $c \in \mathbb{N}$, we construct a constraint system that when solved by a SAT-solver yields an implementation represented as a program tree $\mathcal{T} = (T, \tau)$ with size $|\mathcal{T}| \leq \lfloor c/|\mathcal{A}| \rfloor$ and valid c-bounded annotation functions λ_i for $i \in [n]$ that are witnesses for the implementation. We encode the program tree and the annotation function as uninterpreted functions as explained in the following

We introduce the following variables:

- $L(t,t')$ and $R(t,t')$ for all $t, t' \in T$, iff t' is the left or right child of t, respectively
- $\tau(t,x)$ for all $t \in T$ and $x \in \Sigma$, iff t is labeled with x.
- $\lambda_i^{\mathbb{B}}(t,q,d)$ for all $i \in [n]$, $t \in T$, $q \in Q$ and $d \in \{L, R, D\}$, iff (t,q,d) is reachable in the run graph.
- $\lambda_i^{\#}(t,q,d,x)$ for all $i \in [n]$, $t \in T$, $q \in Q, d \in \{L,R,D\}$ and $0 \leq x < log(c)$, where the annotated number to node (t,q,d) is encoded as a binary number with $log(c)$ many bits. We denote with $\lambda_i^{\#}(t,q,d) \circ k$ for $\circ \in \{<, \leq, =, \geq, >\}$ the relation to some value k or other annotations $\lambda_i^{\#}(t',q',d')$.

The SAT formula $\Phi_{\mathcal{T}}^{\mathcal{A},\triangleright}$ is build based on the following constraints:

- Assure binary tree structure of $\mathcal{T}$, i.e., every node has exactly one parent and the root t_0 has none:

$$\bigwedge_{t' \in T \setminus \{t_0\}} exactlyOne\Big(\Big\{\Big(L(t,t') \wedge \neg R(t,t')\Big) \vee \Big(\neg L(t,t') \wedge R(t,t')\Big) \mid t \in T\Big\}\Big)$$

$$\wedge \bigwedge_{t \in T} \neg L(t,t_0) \wedge \neg R(t,t_0)$$

  where $exactlyOne : X \to \mathbb{B}(X)$ again maps a set to a SAT query to ensure that only one element of the set is $true$ while all remaining are $false$.

- Each node has exactly one label:

$$\bigwedge_{t \in T} exactlyOne(\{\tau(t,x) \mid x \in \Sigma\})$$

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

$$\bigwedge_{i\in[n]} \lambda_i^{\mathbb{B}}(t_0, q_0, d_0) \wedge \bigwedge_{i\in[n],t\in T,q\in Q,d\in\{L,R,D\}} \lambda_i^{\#}(t,q,d) \leq c$$

- All reachable states in the run graph are annotated and fulfill the comparison relation

$$\bigwedge_{i\in[n],(q,t,d)\in\{Q\times T\times\{L,R,D\}\}} \lambda_i^{\mathbb{B}}(q,t,d) \rightarrow \bigwedge_{\sigma\in\Sigma} \tau(t,\sigma)$$
$$\rightarrow \bigwedge_{t'\in T,(d'',d')\in\{(U,L),(U,R),(L,D),(R,D)\}} \mu(t,d'',t',d')$$
$$\rightarrow \bigwedge_{(q',d'')\in\delta_t(q,\sigma,d)} \lambda_i^{\mathbb{B}}(q',t',d') \wedge \lambda_i^{\#}(q,t,d) \triangleright_i^{\mathcal{A}} \lambda_i^{\#}(q',t',d')$$

where $\mu(t,d'',t',d') = \begin{cases} L(t,t') & \text{, if } d''=L \\ R(t,t') & \text{, if } d''=R \\ L(t',t) & \text{, if } d''=U\wedge d=L \\ R(t',t) & \text{, if } d''=U\wedge d=R \end{cases}$

The encoding checks whether universal properties in the run graph hold.

Theorem 3.16
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 $\Phi_{\mathcal{T}}^{\mathcal{A},\triangleright}$ 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 3.14 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 $\lambda_i^{\mathbb{B}}$ be *true* for all reachable states in the run graph $\mathcal{G}$. Then $\Phi_{\mathcal{T}}^{\mathcal{A},\triangleright}$ is satisfied.

" $\Leftarrow$ " :
Let $\Phi_{\mathcal{T}}^{\mathcal{A},\triangleright}$ 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 3.14 the acceptance of $\mathcal{T}$ by $\mathcal{A}$ follows. □

The constructed constraint system is of size $O(|\triangleright| \cdot |T| \cdot |\delta| \cdot |\Sigma|)$ with x many variables where $x \in O(|T| \cdot (|T| + |\Sigma| + |\triangleright| \cdot |Q| \cdot log(|Q| \cdot |T|)))$.

Chapter 4

Synthesis of Reactive Programs

We have already argued that programs as a more succinct representation of implementations are highly desirable. However, in contrast to Mealy machines, that only dependent on the current state map an input to a corresponding output, in programs such a direct mapping is not given. Rather do programs need to be simulated, variables to be altered, expressions to be evaluated and an output statement to be traversed until we produce a corresponding output to a received input, that therefore not only depends on the current position in the code but additionally the *valuation* of all variables.

In [10] reactive program synthesis is solved by means of two-way alternating Büchi tree automata that walk over program trees, while keeping track of the current valuation and the state of a given Büchi specification automaton that is simulated by the input/output received by traversing the program tree. The automaton then accepts a program tree whenever the simulated specification automaton accepts the provided input/output. The constructed automaton is intersected with two more constructed automata that accept syntactically correct programs and reactive programs. Then a reactive and syntactically correct program is synthesized by means of an emptiness check of the obtained automaton.

4.1 Valuations

A valuation is a function that maps from variables to boolean values $s \in S := B \to \mathbb{B}$. Such a valuation represents part of our program state. We can update a valuation with the following notation:

$$\text{for some } b \in B, v \in \mathbb{B} :$$

$$s[b/v](x) = \begin{cases} v & \text{if } b = x \\ s(x) & \text{otherwise} \end{cases}$$

We extend this notation to replace vectors of variables by vectors of values such that we replace a variable at some position i in our variables vector $\vec{b}$ by the respective value v_i of the value vector $\vec{v}$.

$$\text{for some } \vec{b} \in B^n, \vec{v} \in \mathbb{B}^n :$$

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

4.2 Construction of Two-Way Alternating Büchi Tree Automaton

First we translate a given specification into a non-deterministic Büchi word automaton $A_{\overline{spec}}$ as explained in Theorem 2.19. By intuition the automaton we construct guesses an error-path of our program and therefore accepts program trees that do not satisfy the specification. The received input of the environment and the simulation of the underlying specification automaton $A_{\overline{spec}}$ is non-deterministic. We use non-determinism to guess the evaluation of boolean expressions and, for conditionals, check with a universal choice whether the guess was correct.

The constructed two-way alternating Büchi automaton $\mathcal{A}$ over the alphabet Σ_P with the set of states $P^{\mathcal{A}}$,

$$P^{\mathcal{A}}_{expr} = S \times Bool$$

$$P_{exec} = S \times Q \times I \times \{inp, out\} \times Bool$$

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

and initial state $p^{\mathcal{A}}_0 = (s_0, q_0, i_0, inp, 0)$, is defined with the following transitions, where

$s \in S$, $v \in \mathit{Bool}$, $q \in Q$, $i \in I$, $m \in \{\mathit{inp}, \mathit{out}\}$ and $t \in \{0,1\}$.

- **Transitions to evaluate Boolean expressions:**
  - $\delta_{\emptyset}^{\mathcal{A}}((s,1), \mathbf{tt}, D) = \mathit{true}$
    $\delta_{\emptyset}^{\mathcal{A}}((s,0), \mathbf{tt}, D) = \mathit{false}$
  - $\delta_{\emptyset}^{\mathcal{A}}((s,1), \mathbf{ff}, D) = \mathit{false}$
    $\delta_{\emptyset}^{\mathcal{A}}((s,0), \mathbf{ff}, D) = \mathit{true}$
  - $\delta_{\emptyset}^{\mathcal{A}}((s,v), b, D) = \begin{cases} \mathit{true} & \text{, if } s[b] = v \\ \mathit{false} & \text{, otherwise} \end{cases}$
  - $\delta_{LR}^{\mathcal{A}}((s,1), \vee, D) = ((s,1),L) \vee ((s,1),R)$
    $\delta_{LR}^{\mathcal{A}}((s,0), \vee, D) = ((s,0),L) \wedge ((s,0),R)$
  - $\delta_{L}^{\mathcal{A}}((s,v), \neg, D) = ((s,1-v),L)$
- **Transitions to evaluate non I/O statements:**
  - $\delta_{\emptyset}^{\mathcal{A}}((s,q,i,m,t), \mathbf{skip}, D) = ((s,q,i,m,0),U)$
  - $\delta_{L}^{\mathcal{A}}((s,q,i,m,t), \mathit{assign}_b, D) =$
    $\Big(((s[b/0],q,i,m,0),U) \wedge ((s,0),L)\Big) \;\vee\; \Big(((s[b/1],q,i,m,0),U) \wedge ((s,1),L)\Big)$
  - $\delta_{LR}^{\mathcal{A}}((s,q,i,m,t), \mathbf{if}, D) =$
    $\Big(((s,1),L) \wedge ((s,q,i,m,0),LR)\Big) \;\vee\; \Big(((s,0),L) \wedge ((s,q,i,m,0),RR)\Big)$
  - $\delta_{LR}^{\mathcal{A}}((s,q,i,m,t), \mathbf{while}, D)$
    $= \delta_{LR}^{\mathcal{A}}((s,q,i,m,t), \mathbf{while}, R)$
    $= \Big(((s,1),L) \wedge ((s,q,i,m,0),R)\Big) \;\vee\; \Big(((s,0),L) \wedge ((s,q,i,m,0),U)\Big)$
- **Transitions to evaluate input and output:**
  - $\delta_{\emptyset}^{\mathcal{A}}((s,q,i,\mathit{inp},t), \mathbf{input}\ \vec{b}, D) =$
    $\bigvee_{\text{valuations } \mathit{val} \text{ over } \vec{b}} ((s[\vec{b}/\mathit{val}], q, \mathit{val}, \mathit{out}, 0), U)$
  - $\delta_{\emptyset}^{\mathcal{A}}((s,q,i,\mathit{out},t), \mathbf{output}\ \vec{b}, D) = \bigvee_{q' \in \delta_{spec}(q,\vec{i},s[\vec{b}])} ((s,q',i,\mathit{inp},1),U)$
- **Transitions to move to next statement in program:**
  - $\delta_{LR}^{\mathcal{A}}((s,q,i,m,t), \mathbf{;}, D) = ((s,q,i,m,t),L)$
    $\delta_{LR}^{\mathcal{A}}((s,q,i,m,t), \mathbf{;}, L) = ((s,q,i,m,t),R)$
    $\delta_{LR}^{\mathcal{A}}((s,q,i,m,t), \mathbf{;}, R) = ((s,q,i,m,t),U)$

– $\delta^{\mathcal{A}}_{LR}((s,q,i,m,t),\textbf{then},L)$
$= \delta^{\mathcal{A}}_{LR}((s,q,i,m,t),\textbf{then},R)$
$= ((s,q,i,m,t),U)$

– $\delta^{\mathcal{A}}_{LR}((s,q,i,m,t),\textbf{if},R) = ((s,q,i,m,t),U)$

All other transitions evaluate to $false$.

The set of accepting states is defined as

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

The set of states $P^{\mathcal{A}}$ consists of two separate sets for the evaluation of boolean expressions $P^{\mathcal{A}}_{expr}$ and for the normal execution of our program P_{exec}.
Boolean expressions are evaluated by sending a copy of the automaton into the program subtree containing the boolean expression. Boolean evaluations only walk down on the finite subtree and terminate at the bottom to $true$ or $false$, if the evaluation yields the expected result or not, respectively. To evaluate a boolean expression, the automaton requires the current valuation $s \in S$ and the expected evaluation result.
The normal execution states are needed to traverse our program tree and thereby reading input, altering the current valuation and producing output. Such a state consists of the current valuation $s \in S$, the current state of the simulated specification automaton $q \in Q$, the last input read, a flag, whether the next interaction with the environment is an input or an output statement, and a marker for the Büchi acceptance.

The automaton has a Büchi acceptance condition and the set of accepting states is a collection of normal execution states, to be more precise all normal execution states after an output-statement that simulated the underlying specification automaton such that the specification automaton just hit one of its accepting states. This means $\mathcal{A}$ accepts a program tree whenever the underlying automaton accepts the infinite input/output sequence produced by the program tree. Since the specification automaton accepts infinite words that do not satisfy the specification, $\mathcal{A}$ precisely accepts program trees that interact infinitely often with the environment and fail the specification.

For the remainder of this thesis we refer with $\mathcal{A}$ to this constructed two-way alternating Büchi automaton.

4.3 Synthesis

In this section we shortly sketch the synthesis approach introduced in [8]. First we introduce a non-deterministic tree automaton $\mathcal{A}_{pgm}$ that accepts all finite syntactically correct program trees over a fixed set of Boolean variables B with input and output arities $N_{\mathcal{I}}$ and $N_{\mathcal{O}}$, respectively. We then introduce a two-way alternating Büchi tree automaton $\mathcal{A}_{reactive}$ that accepts reactive programs. We then complement $\mathcal{A}$ into a two-way alternating co-Büchi automaton that accepts all program trees that satisfy the given specification. This automaton and $\mathcal{A}_{reactive}$ can then with a exponential blowup be transformed into equivalent non-deterministic tree automata. This transformation is based on the two-way alternating tree automata to non-deterministic tree automata conversion for infinite trees introduced in [10]. The intersection of the three non-deterministic tree automata then accepts reactive programs that do satisfy the specification. The following theorem captures the correctness and complexity of the construction.

Theorem 4.1 ([8])
Let B be a finite set of Boolean variables, and let $N_{\mathcal{I}}, N_{\mathcal{O}} \in N$. Let $\mathcal{A}_{\overline{spec}}$ be a non-deterministic Büchi automaton over the alphabet $\{0,1\}^{N_{\mathcal{I}}+N_{\mathcal{O}}}$. Then we can construct a non-deterministic tree automaton, that precisely accepts the trees corresponding to reactive programs over B and with input/output type $(N_{\mathcal{I}}, N_{\mathcal{O}})$ that on all executions generate input/output sequences that are not in $\mathcal{L}(\mathcal{A})$. Furthermore, this tree automaton can be constructed to be of size $O(exp(|\mathcal{A}_{\overline{spec}}|, exp(B)))$.

Chapter 5

Bounded Synthesis of Reactive Programs

5.1 Automata Construction

In this section, we construct a two-way non-deterministic Büchi tree automaton $\mathcal{B}$ that is equivalent to $\mathcal{A}$ by reconstructing the evaluation of boolean expressions and therefore remove all universal choices. We construct $\mathcal{B}$ without a blowup in the state space, i.e., the size of $\mathcal{B}$ is linearly in $\mathcal{A}$. We then complement $\mathcal{B}$ into a two-way universal co-Büchi tree automaton to match the requirements for the introduced bounded synthesis approach.

5.1.1 Two-way Universal Co-Büchi Tree Automaton

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 we non-deterministically guess whether the expression evaluates to 0 or 1 and then universally send one copy into the boolean expression, that 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. We use this property to reconstruct the boolean evaluation process, such that instead of universally checking whether a boolean expression and a normal execution are accepting, we first deterministically evaluate the boolean expression (note this is done in finitely many steps) and then either continue with the normal execution or reject, if the boolean evaluation yielded the expected result or not.

The only other occurrence of an universal choice is within the boolean evaluation itself, but since we reconstruct this process such that boolean expressions are evaluated deterministically, those choices will be removed.

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

$$P^{\mathcal{B}}_{exec'} = Q \times I \times \{inp, out\}$$

$$P_{exec} = S \times Q \times I \times \{inp, out\} \times Bool$$

$$P^{\mathcal{B}}_{expr} = S \times Bool \times \{\top, \bot\} \times P^{\mathcal{B}}_{exec'}$$

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

and initial state $p^{\mathcal{B}}_0 = (s_0, q_0, i_0, inp, 0)$, is defined with the following transitions, where $s \in S$, $v \in Bool$, $q \in Q$, $i \in I$, $m \in \{inp, out\}$, $t \in \{0, 1\}$, $p \in P^{\mathcal{B}}_{exec'}$ and $r \in \{\top, \bot\}$. Note that transitions different from the ones defined for $\mathcal{A}$ are highlighted by a "**+**"-bullet point.

- **Transitions to evaluate Boolean expressions:**

  + $\delta^{\mathcal{B}}_{\emptyset}((s, 1, \bot, p), \mathbf{tt}, D) = ((s, 1, \top, p), U)$
  $\delta^{\mathcal{B}}_{\emptyset}((s, 0, \bot, p), \mathbf{tt}, D) = ((s, 0, \bot, p), U)$

  + $\delta^{\mathcal{B}}_{\emptyset}((s, 1, \bot, p), \mathbf{ff}, D) = ((s, 1, \bot, p), U)$
  $\delta^{\mathcal{B}}_{\emptyset}((s, 0, \bot, p), \mathbf{ff}, D) = ((s, 0, \top, p), U)$

  + $\delta^{\mathcal{B}}_{\emptyset}((s, v, \bot, p), b, D) = \begin{cases} ((s, v, \top, p), U) & \text{, if } s[b] = v \\ ((s, v, \bot, p), U) & \text{, otherwise} \end{cases}$

  + $\delta^{\mathcal{B}}_{LR}((s, v, \bot, p), \vee, D) = ((s, v, \bot, p), L)$
  $\delta^{\mathcal{B}}_{LR}((s, 1, \top, p), \vee, L) = ((s, 1, \top, p), U)$
  $\delta^{\mathcal{B}}_{LR}((s, 1, \bot, p), \vee, L) = ((s, 1, \bot, p), R)$

  $\delta^{\mathcal{B}}_{LR}((s, 0, \top, p), \vee, L) = ((s, 0, \bot, p), R)$
  $\delta^{\mathcal{B}}_{LR}((s, 0, \bot, p), \vee, L) = ((s, 0, \bot, p), U)$

  $\delta^{\mathcal{B}}_{LR}((s, v, r, p), \vee, R) = ((s, v, r, p), U)$

  + $\delta^{\mathcal{B}}_{L}((s,v,r,p),\neg,D) = ((s,1-v,r,p),L)$
    $\delta^{\mathcal{B}}_{L}((s,v,r,p),\neg,L) = ((s,1-v,r,p),U)$

- **Transitions to evaluate non I/O statements:**

  + $\delta^{\mathcal{B}}_{\emptyset}((s,q,i,m,t),\mathbf{skip},D) = ((s,q,i,m,0),U)$

  + $\delta^{\mathcal{B}}_{L}((s,q,i,m,t),assign_b,D) =$
    $((s,0,\bot,(q,i,m)),L) \;\vee\; ((s,1,\bot,(q,i,m)),L)$
    $\delta^{\mathcal{B}}_{L}((s,v,\top,(q,i,m)),assign_b,L)$
    $= \delta^{\mathcal{B}}_{L}((s,v,\top,(q,i,m)),assign_b,L)$
    $= ((s[b/v],q,i,m,0),U)$

  + $\delta^{\mathcal{B}}_{LR}((s,q,i,m,t),\mathbf{if},D) =$
    $((s,1,\bot,(q,i,m)),L) \;\vee\; ((s,0,\bot,(q,i,m)),L)$
    $\delta^{\mathcal{B}}_{LR}((s,1,\top,(q,i,m)),\mathbf{if},L) = ((s,q,i,m,0),RL)$
    $\delta^{\mathcal{B}}_{LR}((s,0,\top,(q,i,m)),\mathbf{if},L) = ((s,q,i,m,0),RR)$

  + $\delta^{\mathcal{B}}_{LR}((s,q,i,m,t),\mathbf{while},D)$
    $= \delta^{\mathcal{B}}_{LR}((s,q,i,m,t),\mathbf{while},R)$
    $= ((s,1,\bot,(q,i,m)),L) \;\vee\; ((s,0,\bot,(q,i,m)),L)$
    $\delta^{\mathcal{B}}_{LR}((s,1,\top,(q,i,m)),\mathbf{while},L) = ((s,q,i,m,0),R)$
    $\delta^{\mathcal{B}}_{LR}((s,0,\top,(q,i,m)),\mathbf{while},L) = ((s,q,i,m,0),U)$

- **Transitions to evaluate input and output:**

  – $\delta^{\mathcal{B}}_{\emptyset}((s,q,i,inp,t),\mathbf{input}\ \vec{b},D) =$
    $\bigvee_{\text{valuations } val \text{ over } \vec{b}}((s[\vec{b}/val],q,val,out,0),U)$

  – $\delta^{\mathcal{B}}_{\emptyset}((s,q,i,out,t),\mathbf{output}\ \vec{b},D) = \bigvee_{q'\in\delta_{spec}(q,\vec{i},s[\vec{b}])}((s,q',i,inp,1),U)$

- **Transitions to move to next statement in program:**

  – $\delta^{\mathcal{B}}_{LR}((s,q,i,m,t),;,D) = ((s,q,i,m,t),L)$
    $\delta^{\mathcal{B}}_{LR}((s,q,i,m,t),;,L) = ((s,q,i,m,t),R)$
    $\delta^{\mathcal{B}}_{LR}((s,q,i,m,t),;,R) = ((s,q,i,m,t),U)$

  – $\delta^{\mathcal{B}}_{LR}((s,q,i,m,t),\mathbf{then},L)$
    $= \delta^{\mathcal{B}}_{LR}((s,q,i,m,t),\mathbf{then},R)$
    $= ((s,q,i,m,t),U)$

  – $\delta^{\mathcal{B}}_{LR}((s,q,i,m,t),\mathbf{if},R) = ((s,q,i,m,t),U)$

All other transitions evaluate to $false$.

The set of accepting states is defined as

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

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. Additionally, the set of accepting states $F^{\mathcal{A}}$ and $F^{\mathcal{B}}$ are equivalent.

Equivalence of $\mathcal{A}$ and $\mathcal{B}$

In this section we prove that $\mathcal{A}$ and $\mathcal{B}$ accept the exact same program trees. We do so by showing first that $\mathcal{B}$ evaluates boolean expressions to the same result as $\mathcal{A}$ but that instead of terminating with $true$ or $false$ a copy of the automaton reaches the state where the boolean evaluation was started from and this copy contains information of the correct result, that is $\top$ and $\bot$ for $true$ and $false$, respectively. We then use this to show the equivalence of $\mathcal{A}$ and $\mathcal{B}$.

Lemma 5.1
Given a program tree $\mathcal{T}$ with root $t \in T$ that is a boolean expression. If and only if $\mathcal{A}$ in current state $(s, v) \in P^{\mathcal{A}}_{expr}$ send into t evaluates to true/$false$, that is it accepts/rejects $\mathcal{T}$, $\mathcal{B}$ with current state $(s, v, \bot, p) \in P^{\mathcal{B}}_{expr}$ send into t walks over $\mathcal{T}$ and eventually returns from t with the corresponding evaluation result contained in its state (s, v, r, p), where r is $\top/\bot$ if the evaluation result was true/$false$.

Proof. Let $\mathcal{A}$, $\mathcal{B}$ be defined as above, $\mathcal{T}$ be a valid program tree, $t \in T$ be the current input node, that is the root of an boolean expression and $p \in P^{\mathcal{B}}_{exec'}$. Proof by structural induction over boolean expressions.
Base-cases: For $\tau(t)$:

- **tt :**
  - $\delta^{\mathcal{A}}_{\emptyset}\Big((s, 1), \mathbf{tt}, D\Big) = true$
    $\delta^{\mathcal{B}}_{\emptyset}\Big((s, 1, \bot, p), \mathbf{tt}, D\Big) = ((s, 1, \top, p), U)\checkmark$

- $\delta^{\mathcal{A}}_{\emptyset}\Big((s,0),\mathbf{tt},D\Big) = false$
  $\delta^{\mathcal{B}}_{\emptyset}\Big((s,0,\bot,p),\mathbf{tt},D\Big) = ((s,0,\bot,p),U)$✓

- **ff :**
  - $\delta^{\mathcal{A}}_{\emptyset}\Big((s,1),\mathbf{ff},D\Big) = false$
    $\delta^{\mathcal{B}}_{\emptyset}\Big((s,1,\bot,p),\mathbf{ff},D\Big) = ((s,1,\bot,p),U)$✓
  - $\delta^{\mathcal{A}}_{\emptyset}\Big((s,0),\mathbf{ff},D\Big) = true$
    $\delta^{\mathcal{B}}_{\emptyset}\Big((s,0,\bot,p),\mathbf{ff},D\Big) = ((s,0,\top,p),U)$✓

- ***b* :**

$$\delta^{\mathcal{A}}_{\emptyset}((s,v),b,D) = \begin{cases} true & \text{, if } s[b]=v \\ false & \text{, otherwise} \end{cases}$$

$$\delta^{\mathcal{B}}_{\emptyset}((s,v,\bot,p),b,D) = \begin{cases} ((s,v,\top,p),U)✓ & \text{, if } s[b]=v \\ ((s,v,\bot,p),U)✓ & \text{, otherwise} \end{cases}$$

Inductive-step: For $\tau(t) =$

- **$\neg$:**
  $\delta^{\mathcal{A}}_{L}((s,v),\neg,D) = ((s,1-v),L)$
  By induction $\delta^{\mathcal{B}}_{L}((s,v,\bot,p),\neg,D) = ((s,1-v,r,p),L)$ eventually returns with $(s,1-v,r,p)$ where r is $\top/\bot$ if $((s,1-v),L)$ evaluates to $true/false$, respectively.
  $\delta^{\mathcal{B}}_{L}((s,1-v,r,p),\neg,L) = ((s,v,r,p),U)$✓

- **$\vee$:**
  - $\delta^{\mathcal{A}}_{LR}((s,1),\vee,D) = ((s,1),L) \vee ((s,1),R)$ evaluates to either
    - *true* meaning $((s,1),L)$ or $((s,1),R)$ evaluated to *true*.
      - For ((s,1),L) evaluating to *true* by induction we have $\delta^{\mathcal{B}}_{LR}((s,1,\bot,p),\vee,D) = ((s,1,\bot,p),L)$ eventually returning with $(s,1,\top,p)$ from the left subtree and by definition $\delta^{\mathcal{B}}_{LR}((s,1,\top,p),\vee,L) = ((s,1,\top,p),U)$.✓
      - For ((s,1),L) evaluating to *false* by induction we have $(s,1,\bot,p)$ eventually returning from the left subtree and therefore $\delta^{\mathcal{B}}_{LR}((s,1,\bot,p),\vee,L) = ((s,1,\bot,p),R)$. With ((s,1),R) being *true* we have by induction $(s,1,\top,p)$ returning from the right subtree and therefore $\delta^{\mathcal{B}}_{LR}((s,1,\top,p),\vee,R) = ((s,1,\top,p),U)$ by definition.✓

    * or $false$ meaning both $((s,1),L)$ and $((s,1),R)$ evaluated to $false$. By induction we have $(s,1,\bot,p)$ eventually returning from the left subtree and therefor $\delta^{\mathcal{B}}_{LR}((s,1,\bot,p),\vee,L) = ((s,1,\bot,p),R)$. With induction on $((s,1),R)$ we have $(s,1,\bot,p)$ returning from the right subtree and by definition follows $\delta^{\mathcal{B}}_{LR}((s,v,\bot,p),\vee,R) = ((s,v,\bot,p),U)$.✓

  - $\delta^{\mathcal{A}}_{LR}((s,0),\vee,D) = ((s,0),L) \wedge ((s,0),R)$ evaluates to either

    * $true$ meaning both $((s,0),L)$ and $((s,0),R)$ evaluating to $true$. By induction on $((s,0),L)$ we have $\delta^{\mathcal{B}}_{LR}((s,0,\bot,p),\vee,D) = ((s,0,\bot,p),L)$ eventually returning with $(s,0,\top,p)$ from the left subtree. By definition we have $\delta^{\mathcal{B}}_{LR}((s,0,\top,p),\vee,L) = ((s,0,\bot,p),R)$ that by induction on $((s,0),R)$ eventually returns with $(s,0,\top,p)$ from the right subtree and by definition $\delta^{\mathcal{B}}_{LR}((s,0,\top,p),\vee,R) = ((s,0,\top,p),U)$.✓

    * or $false$ meaning either $((s,0),L)$ or $((s,0),R)$ evaluating to $false$.

      · For $((s,0),L)$ evaluating to $false$. By induction on $((s,0),L)$ we have $\delta^{\mathcal{B}}_{LR}((s,0,\bot,p),\vee,D) = ((s,0,\bot,p),L)$ eventually returning with $(s,0,\bot,p)$ from the left subtree and therefore by definition $\delta^{\mathcal{B}}_{LR}((s,0,\bot,p),\vee,L) = ((s,0,\bot,p),U)$.✓

      · For $((s,0),L)$ and $((s,0),R)$ evaluating to $true$ and $false$, respectively. By induction on $((s,0),L)$ we have $\delta^{\mathcal{B}}_{LR}((s,0,\bot,p),\vee,D) = ((s,0,\bot,p),L)$ eventually returning with $(s,0,\top,p)$ from the left subtree and therefore $\delta^{\mathcal{B}}_{LR}((s,0,\top,p),\vee,L) = ((s,0,\bot,p),R)$ by induction on $((s,0),R)$ eventually returning with $(s,0,\bot,p)$ from the right subtree. By definition $\delta^{\mathcal{B}}_{LR}((s,0,\bot,p),\vee,R) = ((s,0,\bot,p),U)$.✓

□

We now proof that besides boolean expressions both automata traverse program trees in the same way, more precisely, they traverse the same states of P_{exec} and since the accepting states $F^{\mathcal{A}} = F^{\mathcal{B}} \subset P_{exec}$ they accept the same trees.

Theorem 5.2
$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

Proof. " $\Rightarrow$ " :
Let $\mathcal{T} \in \mathcal{L}(\mathcal{A})$ be a program tree with accepting run tree $\mathcal{R}_{\mathcal{A}} = \langle R_{\mathcal{A}}, \tau_{\mathcal{A}}\rangle$. The initial

state p_0 of $\mathcal{A}$ is a normal execution state ($p_0 \in P_{exec}$). The transitions of $\mathcal{A}$ are defined such that whenever $\mathcal{A}$ is in a normal execution state $p \in P_{exec}$ exactly one successor is again a normal execution state $p' \in P_{exec}$. There can be an additional successor state that is used to evaluate a boolean expression but those subtrees of the accepting run graph are finite and terminate to *true* on every path. Therefore there only exists a single infinite path

$$r_0 r_1 \ldots = (t_0, p_0, d_0)(t_1, p_1, d_1) \ldots \in \Big(T \times P_{exec} \times \{L, R, D\}\Big)^{\omega}$$

in the accepting run tree $\mathcal{R}_{\mathcal{A}}$ that satisfies the Büchi acceptance condition.

We show that $\mathcal{B}$ visits the same states $p_0 p_1 \ldots \in P_{exec}^{\omega}$ in the same order while traversing $\mathcal{T}$, that is whenever $\mathcal{A}$ is in state $p_i \in P_{exec}$ reading a node $t_i \in \mathcal{T}$ moving to node t_{i+1} and state p_{i+1}, $\mathcal{B}$ in state p_i reading node t_i eventually moves to node t_{i+1} with state p_{i+1}. We show this with a case analysis over possible inputs $\tau_{\mathcal{A}}(t_i)$:

- $assign_b$: In this case r_i has two children r_{i+1} and (t', p', L) with $p' = (s, v) \in P_{expr}^{\mathcal{A}}$. Since the run tree is accepting, the boolean evaluation evaluates to *true*. For $\mathcal{B}$ we choose $((s, v, \bot, (q, i, m)), L)$ as valid move. With Lemma 5.1 we know that $\mathcal{B}$ traverses the boolean expression subtree and returns with state $(s, v, \top, (q, i, m))$ to t_i. The only valid transition for $\mathcal{B}$ now is $((s[b/v], q, i, m, 0), U)$, that is a move to t_{i+1} with state p_{i+1}.✓

- **if**: In this case r_i has two children r_{i+1} and (t', p', d') with $p' = (s, v) \in P_{expr}^{\mathcal{A}}$. Since the run tree is accepting, the boolean evaluation evaluates to *true*. For $\mathcal{B}$ we choose $((s, v, \bot, (q, i, m)), L)$ as valid move. With Lemma 5.1 we know that $\mathcal{B}$ traverses the boolean expression subtree and returns with state $(s, v, \top, (q, i, m))$ to t_i. Depending on v having the value 0 or 1 the only valid transition for $\mathcal{B}$ is either $((s, q, i, m, 0), RR)$ or $((s, q, i, m, 0), RL)$, respectively. Either way it matches the move of $\mathcal{A}$ and therefore is a move to t_{i+1} with state p_{i+1}.✓

- **while**: In this case r_i has two children r_{i+1} and (t', p', d') with $p' = (s, v) \in P_{expr}^{\mathcal{A}}$. Since the run tree is accepting, the boolean evaluation evaluates to *true*. For $\mathcal{B}$ we choose $((s, v, \bot, (q, i, m)), L)$ as valid move. With Lemma 5.1 we know that $\mathcal{B}$ traverses the boolean expression subtree and returns with state $(s, v, \top, (q, i, m))$ to t_i. Depending on v having the value 0 or 1 the only valid transition for $\mathcal{B}$ is either $((s, q, i, m, 0), U)$ or $((s, q, i, m, 0), R)$, respectively. Either way it matches the move of $\mathcal{A}$ and therefore is a move to t_{i+1} with state p_{i+1}.✓

All other valid transitions are equally defined such that $\mathcal{B}$ can make the same choices.

Since $\mathcal{B}$ visits for every accepted run tree the same normal execution states equally often and the set of accepting states for $\mathcal{A}$ and $\mathcal{B}$ are equivalent and a subset of normal execution states $\mathcal{B}$ accepts the same program trees as $\mathcal{A}$.

" $\Leftarrow$ " :
Let $\mathcal{T} \in \mathcal{L}(\mathcal{B})$ be a program tree with accepting run tree $\mathcal{R}_\mathcal{B} = \langle R_\mathcal{B}, \tau_\mathcal{B} \rangle$. The initial state p_0 of $\mathcal{B}$ is a normal execution state ($p_0 \in P_{exec}$). Since $\mathcal{B}$ has no universal choices the run tree is 1-ary, that is a single infinite path

$$r_0 r_1 \ldots = (t_0, p_0, d_0)(t_1, p_1, d_1) \ldots \in \Big(T \times P^\mathcal{B} \times \{L, R, D\}\Big)^\omega$$

The infinite sequence of states $p_0 p_1 \ldots$ hits the set of accepting states $F^\mathcal{B}$ infinitely often and is structured as follows:

- $p_0 \in P_{exec}$
- $\forall i : p_i \in P_{exec} \to p_{i+1} \in P_{exec} \vee \Big(\exists j : p_{i+1} p_{i+2} \ldots p_{i+j} \in (P^\mathcal{B}_{expr})^* \wedge p_{i+j+1} \in P_{exec}\Big)$

Which means the sequence of states starts with a normal execution state and contains infinitely many.

We show that $\mathcal{A}$ can visit the same normal execution states as $\mathcal{B}$, more precisely: Let $\mathcal{B}$ be in state $p_i \in P_{exec}$ reading a node $t_i \in \mathcal{T}$. If $\mathcal{B}$ moves to node t_{i+1} with state $p_{i+1} \in P_{exec}$, then $\mathcal{A}$ can also perform a valid move from t_i with state p_i to t_{i+1} with state p_{i+1}.
If $\mathcal{B}$ moves to a state $p_{i+1} \in P^\mathcal{B}_{expr}$ and traverses the boolean expression subtree and therefore visits the states $p_{i+2} p_{i+3} \ldots p_{i+j-1} \in (P^\mathcal{B}_{expr})^*$ until it returns to node $t_i (= t_{i+j})$ with state $p_{i+j} \in P^\mathcal{B}_{expr}$ and subsequently moves to node t_{i+j+1} with state $p_{i+j+1} \in P_{exec}$, then $\mathcal{A}$ can also perform a valid move from t_i with state p_i to node t_{i+j+1} with state p_{i+j+1}.

In the first case, where $\mathcal{B}$ moves from normal execution state to another normal execution state $\mathcal{A}$ can always do the same move since those transitions are equally defined. We show the second case with a case analysis over the possible labels $\tau_\mathcal{B}(t_i)$. Let $p_i = (s, q, i, m, t) \in P_{exec}$, $t_i \in \mathcal{T}$, $p_{i+1} = (s, v, \bot, (q, i, m)) \in P^\mathcal{B}_{expr}$ and $p_{i+j} = (s, v, \top, (q, i, m))$ (since the run tree is valid the boolean evaluation has to be successful).

- $assign_b$: In this case $\mathcal{B}$ moves from p_{i+j} to $(s[b/v], q, i, m, 0)$. Depending on v, $\mathcal{A}$ moves from t_i with state p_i into t_{i+j+1} with state $(s[b/v], q, i, m, 0)$ and into t_{i+1}

with state (s, v). The second copy send into the boolean expression terminates to *true* based on Lemma 5.1.✓

- **if** and **while**: $\mathcal{A}$ moves from t_i with state p_i into t_{i+j+1} with state p_{i+j+1} and depending on v also into t_{i+1} with state (s, v). Again with Lemma 5.1 the second copy send into the boolean expression terminates to *true*.✓

Since $\mathcal{A}$ visits the same normal execution states as $\mathcal{B}$ in the same order and the set of accepting states are equivalent and is a subset of normal execution states, $\mathcal{A}$ fulfills the Büchi acceptance condition. □

We now complement the constructed two-way non-deterministic Büchi automaton to a two-way universal co-Büchi automaton. For the rest of this thesis 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 only finitely often with the environment. How our approach guarantees reactiveness is introduced in Section 5.1.3. First we guarantee that all programs are syntactically correct.

5.1.2 Guarantee Syntactically Correctness

$\mathcal{A}_{pgm}$ as introduced in Section 4.3 is a non-deterministic tree automaton that checks for correct syntax. One can construct $\mathcal{A}_{pgm}$ as a two-way universal co-Büchi automaton $\mathcal{B}_{pgm}$, that universally checks each branch by walking down on the tree and terminating at the leaves.

We define $\mathcal{B}_{pgm} = (\Sigma_P, \{\text{STMT}, \text{EXPR}, \text{THEN}\}, \text{STMT}, \delta_L, \delta_R, \delta_{LR}, \delta_\emptyset, \text{CO-BÜCHI}(\emptyset))$, where the transition function is defined by:

- $\delta_{LR}(\text{STMT}, ;, D) = (\text{STMT}, L) \wedge (\text{STMT}, R)$
- $\delta_{LR}(\text{STMT}, \textbf{if}, D) = (\text{EXPR}, L) \wedge (\text{THEN}, R)$
- $\delta_{LR}(\text{THEN}, \textbf{then}, D) = (\text{STMT}, L) \wedge (\text{STMT}, R)$
- $\delta_{LR}(\text{STMT}, \textbf{while}, D) = (\text{EXPR}, L) \wedge (\text{STMT}, R)$
- $\delta_L(\text{STMT}, assign_b, D) = (\text{EXPR}, L)$

- $\delta_\emptyset(\text{STMT}, \textbf{skip}, D)$
  $= \delta_\emptyset(\text{STMT}, \textbf{input } \vec{b}, D)$
  $= \delta_\emptyset(\text{STMT}, \textbf{output } \vec{b}, D) = true$

- $\delta_{LR}(\text{EXPR}, \vee, D) = (\text{EXPR}, L) \wedge (\text{EXPR}, R)$

- $\delta_L(\text{EXPR}, \neg, D) = (\text{EXPR}, L)$

- $\delta_\emptyset(\text{EXPR}, \textbf{tt}, D)$
  $= \delta_\emptyset(\text{EXPR}, \textbf{ff}, D)$
  $= \delta_\emptyset(\text{EXPR}, b, D) = true$

Note that $\mathcal{B}$ can be easily intersected with $\mathcal{B}_{pgm}$ due to the universal branching mode, that is one can simulate both automata simultaneously simply by sending one copy into the corresponding $\mathcal{B}_{pgm}$-states from the initial state of $\mathcal{B}$.

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 of our program tree that are unreachable. It is now arguable whether the intersection with $\mathcal{B}_{pgm}$ 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 program. Therefor, we continue with $\mathcal{B}$ not intersected with $\mathcal{B}_{pgm}$.

5.1.3 Guarantee Reactiveness

It now remains to guarantee reactiveness of the programs accepted by $\mathcal{B}$. We introduce $\mathcal{B}_{reactive}$, that is a two-way universal Büchi automaton that only accepts program trees that are reactive. This automaton is designed with the exact same states and transitions as $\mathcal{B}$ only with another acceptance condition. The intersection of $\mathcal{B}$ and $\mathcal{B}_{reactive}$ then yields a two-way universal Streett automaton we use to build our constraint system for bounded synthesis.

We construct a two-way universal Büchi automaton $\mathcal{B}_{reactive}$ based on $\mathcal{B}$ that accepts reactive program trees. Formally, $\mathcal{B}_{reactive} = (\Sigma_P, P^{\mathcal{B}}, \delta_L^{\mathcal{B}}, \delta_R^{\mathcal{B}}, \delta_{LR}^{\mathcal{B}}, \delta_{\emptyset}^{\mathcal{B}}, \text{BÜCHI}(F_{reactive}^{\mathcal{B}}))$ is a tuple that only differs from $\mathcal{B}$ in the acceptance condition. The set of accepting states is defined as:

$$F_{reactive}^{\mathcal{B}} = \Big\{(s, q, i, m, 1) \mid \forall s, q, i, m\Big\}$$

So $\mathcal{B}_{reactive}$ informally accepts a program tree if 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.

We now intersect $\mathcal{B}$ and $\mathcal{B}_{reactive}$ into a two-way universal Streett automaton $\mathcal{B}'$. Formally, $\mathcal{B}' = (\Sigma_P, P^{\mathcal{B}}, \delta_L^{\mathcal{B}}, \delta_R^{\mathcal{B}}, \delta_{LR}^{\mathcal{B}}, \delta_{\emptyset}^{\mathcal{B}}, \text{STREETT}(F^{\mathcal{B}'}))$, where

$$F^{\mathcal{B}'} = \Big\{(F^{\mathcal{B}}, \emptyset), (P^{\mathcal{B}}, F_{reactive}^{\mathcal{B}})\Big\}$$

Theorem 5.3
$\mathcal{L}(\mathcal{B}') = \mathcal{L}(\mathcal{B}) \cap \mathcal{L}(\mathcal{B}_{reactive})$

Proof. " $\Rightarrow$ " : Let $\mathcal{T} \in \mathcal{L}(\mathcal{B}')$ be an arbitrary program tree. Let π be an arbitrary infinite path in the run tree of $\mathcal{B}'$ and $\mathcal{T}$. Since $\mathcal{B}'$, $\mathcal{B}_{reactive}$ and $\mathcal{B}$ have the same states and transitions the path π also exists in the run tree of $\mathcal{B}$ and $\mathcal{T}$ and in the run tree of $\mathcal{B}_{reactive}$ and $\mathcal{T}$. Then for every tuple $(A, G) \in F^{\mathcal{B}'}$ either A is hit finitely often or G is hit infinitely often. For $(F^{\mathcal{B}}, \emptyset) \in F^{\mathcal{B}'}$ it holds that $\emptyset$ cannot be hit. Then $F^{\mathcal{B}}$ is only hit finitely often and therefore $\mathcal{T} \in \mathcal{L}(\mathcal{B})$. For $(P^{\mathcal{B}}, F_{reactive}^{\mathcal{B}}) \in F^{\mathcal{B}'}$ it holds that $P^{\mathcal{B}}$ is hit infinitely often since π is infinite. Then $F_{reactive}^{\mathcal{B}}$ is hit infinitely often and therefore $\mathcal{T} \in \mathcal{L}(\mathcal{B}_{reactive})$.

" $\Leftarrow$ " : Let $\mathcal{T} \in \mathcal{L}(\mathcal{B}) \cap \mathcal{L}(\mathcal{B}_{reactive})$ be an arbitrary program tree. Let π be an arbitrary infinite path in the run tree of $\mathcal{B}$ and $\mathcal{T}$ or $\mathcal{B}_{reactive}$ and $\mathcal{T}$. Since the states and transitions are the same the path also exists in the respective other run tree. Since $\mathcal{B}'$, $\mathcal{B}_{reactive}$ and $\mathcal{B}$ have the same states and transitions the path π also exists in the run tree of $\mathcal{B}'$ and $\mathcal{T}$. Since $\mathcal{T} \in \mathcal{L}(\mathcal{B})$ the path π visits $F^{\mathcal{B}}$ only finitely often. Therefore, $(F^{\mathcal{B}}, \emptyset) \in F^{\mathcal{B}'}$ is satisfied. Since $\mathcal{T} \in \mathcal{L}(\mathcal{B}_{reactive})$ the path π visits $F_{reactive}^{\mathcal{B}}$ infinitely often. Therefore, $(P^{\mathcal{B}}, F_{reactive}^{\mathcal{B}}) \in F^{\mathcal{B}'}$ is satisfied. $\square$

5.2 Bounded Synthesis

We now can by means of the encoding $\Phi_S^{\mathcal{B}'}$ introduced in Section 3.3 synthesize program trees accepted by $\mathcal{B}'$, i.e., precisely program trees that correspond to reactive programs that satisfy the given specification.

Theorem 5.4
Let B be a finite set of Boolean variables and let $N^{\mathcal{I}}, N^{\mathcal{O}} \in \mathbb{N}$. For a specification given as an LTL formula φ over the alphabet $\{0,1\}^{N^{\mathcal{I}}+N^{\mathcal{O}}}$, we can construct a two-way universal Streett automaton $\mathcal{B}'$ that precisely accepts the trees corresponding to reactive programs over B and with input/output type $(N^{\mathcal{I}}, N^{\mathcal{O}})$ that on all executions generate I/O sequences that are accepted by the specification automaton.

Furthermore we construct a SAT encoding $\Phi_S^{\mathcal{B}'}$ that is satisfiable for a given bound $c \in \mathbb{N}$, iff there exists a program tree $\mathcal{T}$ with size $|\mathcal{T}| \leq \lfloor c/|\mathcal{B}'| \rfloor$ accepted by $\mathcal{B}'$.

Size of construction

Note that the number of boolean variables $|B|$ dominates $N^{\mathcal{I}}$ and $N^{\mathcal{O}}$. 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, that is exponential in the size of the specification formula as discussed in Section 2.3.1. The constructed constraint system $\Phi_S^{\mathcal{B}'}$ is of size $O(|T| \cdot |\delta| \cdot |\Sigma_P|)$ with x many variables, where $x \in O(|T| \cdot (|T| + |\Sigma_P| + |Q| \cdot log(|Q| \cdot |T|)))$. Note that $|\Sigma_P| \in O(|B|^{N_{\mathcal{I}}+N_{\mathcal{O}}})$ grows polynomial in the number of variables for fixed input/output arities.

Chapter 6

Conclusion and Further Work

We introduced a generalized approach to bounded synthesis that is applicable whenever all possible runs of an 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 by an automaton 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 can then be encoded as a SAT query that is satisfiable if and only if there exists an implementation satisfying a given size bound that is accepted by the automaton.

We constructed for a specification given as an LTL-formula a two-way universal Streett automaton that accepts reactive programs that satisfy the given specification. We then constructed a run graph thats represents all possible runs and applied the generalized bounded synthesis approach. On these grounds we could construct a SAT query that guesses a reactive program of bounded size as well as valid annotation functions that witness the correctness of the synthesized program.

Program Repair

The presented approach is also applicable to program repair. When for example a hole in a program needs to be filled such that the overall program fulfills a given specification, one fixes the already provided program in the constraint system while the remaining subtree can then be synthesized by solving the constraint system. Previous approaches like game based solutions were introduced in [4, 5], but our approach improves in terms of simplicity.

Variable Bound

While fixing the set of variables for our reactive programs we implicitly introduced a bound on the amount of used variables. Intuitively, this bound on the variables correlates with the size of the synthesized program. For example could repeated calculations be replaced by a single calculation and then whenever needed only the variable is accessed instead of a recalculation. On the other hand, our constructed automaton grows exponential in the amount of variables as well as the to be solved constraint system. The determination of an appropriate amount of variables that justifies this trade-of between runtime and probably smaller and structurally simpler programs is left open as further work.

Bound on Output-Distance

To ensure reactiveness we introduced a two-way universal Büchi tree automata $\mathcal{B}_{reactive}$ that accepts reactive programs and intersected this automata with the constructed two-way universal co-Büchi tree automata $\mathcal{B}$, that accepts programs that satisfy the specification. The resulting automata then was an intersection of these two automata, i.e., a two-way universal Streett tree automata.

Instead of the introduction of $\mathcal{B}_{reactive}$ one could also bound the maximal distance between two output statements in the run graph. Since the program alternates between input and output statements and after every output statement another output statement needs to be traversed in finitely many steps, this bound would ensure reactiveness of the implementation. Intuitively, this resembles the Büchi condition of $\mathcal{B}_{reactive}$, that also requires only finitely many steps between two output statements. However, instead of intersecting the two automata into an automaton with Streett acceptance, we keep the Büchi and co-Büchi acceptance conditions and introduce two independently bounded annotation functions to express the acceptance conditions, respectively. This bound then correlates with the worst-case response-time of the implementation measured in evaluation steps in the corresponding program tree and represents an additional output-sensitive property that can be ensured with bounded synthesis.

List of Definitions

References

[1] Bernd Finkbeiner and Felix Klein. "Bounded Cycle Synthesis." English. In: ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9779. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016. ISBN: 978-3-319-41528-4. DOI: 10.1007/978-3-319-41528-4. URL: http://dx.doi.org/10.1007/978-3-319-41528-4.

[2] Bernd Finkbeiner and Sven Schewe. "Bounded synthesis." English. In: *International Journal on Software Tools for Technology Transfer* 15.5-6 (2013), pp. 519–539. ISSN: 1433-2779. DOI: 10.1007/s10009-012-0228-z. URL: http://dx.doi.org/10.1007/s10009-012-0228-z.

[3] Joyce Friedman. "Church Alonzo. Application of recursive arithmetic to the problem of circuit synthesisSummaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton, N. J., 1960, pp. 3–50. 3a-45a." In: *Journal of Symbolic Logic* 28.4 (1963), pp. 289–290. DOI: 10.2307/2271310.

[4] Andreas Griesmayer, Roderick Bloem, and Byron Cook. "Repair of Boolean Programs with an Application to C." In: *Proceedings of the 18th International Conference on Computer Aided Verification.* CAV'06. Seattle, WA: Springer-Verlag, 2006, pp. 358–371. ISBN: 3-540-37406-X, 978-3-540-37406-0. DOI: 10.1007/11817963_33. URL: http://dx.doi.org/10.1007/11817963_33.

[5] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. "Program Repair as a Game." In: *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings.* Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 226–238. ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_23. URL: https://doi.org/10.1007/11513988_23.

[6] Ayrat Khalimov and Roderick Bloem. “Bounded Synthesis for Streett, Rabin, and *CTL**.” In: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II.* Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 333–352. ISBN: 978-3-319-63390-9. DOI: `10.1007/978-3-319-63390-9_18`. URL: `https://doi.org/10.1007/978-3-319-63390-9_18`.

[7] Orna Kupferman and Moshe Y. Vardi. “Safraless Decision Procedures.” In: *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science.* FOCS ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 531–542. ISBN: 0-7695-2468-0. DOI: `10.1109/SFCS.2005.66`. URL: `https://doi.org/10.1109/SFCS.2005.66`.

[8] Parthasarathy Madhusudan. *Synthesizing Reactive Programs.* 2011. DOI: `10.4230/LIPIcs.CSL.2011.428`. URL: `https://doi.org/10.4230/LIPIcs.CSL.2011.428`.

[9] Amir Pnueli. “The Temporal Logic of Programs.” In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* SFCS ’77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`. URL: `http://dx.doi.org/10.1109/SFCS.1977.32`.

[10] Moshe Y. Vardi. “Reasoning About The Past with Two-Way Automata.” In: *Proceedings of the 25th International Colloquium on Automata, Languages and Programming.* ICALP ’98. London, UK, UK: Springer-Verlag, 1998, pp. 628–641. ISBN: 3-540-64781-3. URL: `http://dl.acm.org/citation.cfm?id=646252.685998`.

[11] M.Y. Vardi and P. Wolper. “Reasoning about Infinite Computations.” In: *Information and Computation* 115.1 (1994), pp. 1–37. ISSN: 0890-5401. DOI: `https://doi.org/10.1006/inco.1994.1092`. URL: `http://www.sciencedirect.com/science/article/pii/S0890540184710923`.