Master Thesis

Synthesis of Distributed Reactive Programs

Heinrich Ody

October 31, 2013

Supervisors

Prof. Bernd Finkbeiner, Ph.D.

Markus Rabe, M.Sc.

Examiners

Prof. Bernd Finkbeiner, Ph.D.

Prof. Ruzica Piskac, Ph.D.

Eidesstattliche Erklärung Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig 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.

Einverständisserklärung 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, _____ _____

Acknowledgements

Most importantly, I would like to thank my friends Bayram Kiran, Krishna Narasimhan and Zhazira Oskenbayeva for the countless coffee breaks we had together! Further, I am grateful for the nice and quiet working sessions we shared.

I thank my supervisors Prof. Bernd Finkbeiner and Markus Rabe for their patience when I would not understand something during our discussions and for the time they had for me.

And I thank Sören Jeserich for the discussions and tips on formal definitions.

At last, I thank my girlfriend Ludmila Tsesarska for her support during my master thesis and my whole master studies.

Abstract

We consider the automatic synthesis of a reactive implementation from a temporal specification. Current synthesis approaches usually create a transition system, which may then be transformed into another representation such as a circuit or a program. However, often a transition system is not a desirable intermediate product e.g. because it may be too large. Reactive program synthesis directly extracts a program from the specification, and avoids transition systems as intermediate result. In this thesis we investigate the distributed synthesis of reactive programs. We prove the undecidability of the general synthesis of distributed reactive programs and we argue why the decidability results of the classical distributed synthesis probably do not hold for distributed program synthesis. Also, we prove that for a restricted programming language, the synthesis of distributed programs is decidable for all architectures. For this programming language we provide a synthesis procedure based on ω-tree automata.

Contents

1 Introduction

Given a temporal specification, reactive synthesis is the automatic extraction of an implementation from the specification s.t. the specification is never violated by the implementation. In this case we say that the implementation *realizes* the specification.

Most of the current approaches to reactive synthesis create transition systems. We refer to these approaches as classical reactive synthesis. However, usually we prefer a symbolic result over an explicit one such as a transition system. We are interested in a circuit or a program or another symbolic representation, because the goal of synthesis is to create implementations that can be deployed. Further, a program is more accessible to human inspection and we can use existing tools like compilers etc. to optimize a synthesized program. Recently, the synthesis of reactive programs has been explored by Madhusudan [10]. Madhusudan defines a simple programming language, which is parametrized with the amount of variables it can use. Then, for a given specification he synthesizes a program realizing the specification, if there exists one. He gives the following motivating example: For a fixed set S we want to synthesize an implementation that controls a subset $S' \subseteq S$ to which the environment may

- add or remove elements from S and
- where the environment can query the membership of an element in S'.

Let $|S| = 50$, then a program realizing the specification could have 50 boolean variables that indicate the membership in S' and about $n \log n \approx 150$ lines of code. A transition system would have 2^{50} states, which is not useful. The program however can be deployed and if necessary, read by humans.

In this thesis we investigate the *synthesis of distributed reactive programs*, where we assume the communication to be synchronized. An architecture defines the processes, the communication between them and the memory processes may use. In Figure 1.1 we show an architecture with two processes. P_1 receives one input from the environment, outputs a vector of three signals to the environment and has one bit of memory available. P_2 receives three inputs, outputs one value and has three bits of memory. Then in the synthesis of

Figure 1.1: An example architecture

distributed reactive programs we are given a specification and an architecture and we want to find programs s.t. their joint behavior realizes the specification over the architecture.

Assume we are given the architecture in Figure 1.1 and the LTL specification

$$\square\Big(\overrightarrow{O_1}[0] \iff \overrightarrow{I_2}[0] \wedge \overrightarrow{O_1}[1] \iff \overrightarrow{I_2}[1]$$
$$\vee\, \overrightarrow{O_1}[1] \iff I_2[1] \wedge O_1[2] \iff \overrightarrow{I_2}[2]$$
$$\vee\, \overrightarrow{O_1}[2] \iff \overrightarrow{I_2}[2] \wedge \overrightarrow{O_1}[0] \iff \overrightarrow{I_2}[0]\Big) \implies \square(I_1 \iff \bigcirc\bigcirc O_2)$$

which demands: If in the input vector of P_1 and in the output vector of P_2 there is always at most one index i s.t. $O_1[i]$ *XOR* $I_2[i]$, then O_2 will always have the value I_1 had two steps ago. This is a form of error correction by redundancy. By letting the communication go over the environment we allow the environment to introduce an error in the communication, which we bound in the specification. Note that we compare O_2 to the value of from two steps ago because we assume a delay in the communication. A distributed program realizing the specification over the architecture is shown in Figure 1.2. The goal of the synthesis of distributed reactive programs is to find such a distributed program automatically.

```
while tt
    input b
    output b, b, b
```

```
while tt
    input b_0, b_1, b_2
    b_0 := b_0 ∧ b_1 ∨ b_1 ∧ b_2 ∨ b_2 ∧ b_0
    output b_0
```

Figure 1.2: A distributed program, where the left program implements P_1 and the right program implements P_2

We prove the undecidability of the synthesis of distributed reactive programs for a particular architecture. This undecidability is not surprising, because the classical synthesis of distributed synthesis is undecidable as well [13]. Further, we conjecture the undecidability of the synthesis of distributed reactive programs over hierarchical architectures for which the classical synthesis of distributed synthesis is decidable [13, 9]. Also, we define a programming language which is parametrized in the number of output statements a program is allowed to use. For this programming language we solve the synthesis of distributed reactive programs for all architectures. At last, we implemented our approach as a proof of concept.

Related Work We strongly depend on the results of the synthesis of sequential reactive programs [10]. Finkbeiner and Schewe define an exact characterization of architectures, for which the classical distributed synthesis problem is always decidable [6]. Also, the authors complete previous results on the synthesis of distributed systems [9, 13] and solve it for all decidable architectures. The case of asynchronous communication also has been considered [14, 16]. Brutsch simplified the results from Madhusudan [2].

2 Preliminaries

In this section we define terms we use throughout this thesis. We define different kinds of automata, the synthesis of transition systems and the synthesis of reactive programs.

2.1 Automata

In this section we introduce ω-automata, which are automata that run on infinite inputs. First we define ω-automata over infinite words. They were first introduced by Büchi in [3] to show the decidability of S1S. Then we define automata over infinite trees. Tree automata were first considered by [15] to show decidability of S2S. For a deeper introduction to word- and tree automata see [17]. The languages recognized by these automata are called ω-regular word- and tree languages.

2.1.1 Automata on Infinite Words

First we introduce nondeterministic Büchi automata and nondeterministic co-Büchi automata. Then we generalize these automata to parity automata. At the end we also introduce universal automata and generalize the automata introduced to alternating automata.

Nondeterministic Büchi Word Automata A nondeterministic Büchi word automaton over infinite words is a tuple $A = (\Sigma, Q, \delta, I, F)$ where Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, I is the set of initial states and $F \subseteq Q$ the set of accepting or final states. Infinite words will be written as $w = \sigma_0\sigma_1 \ldots$, where $\sigma_i \in \Sigma$. A run of an automaton A on the word w is an infinite sequence of states $r = s_0 s_1 \ldots$ s.t. $s_0 \in I$ and $s_{i+1} \in \delta(s_i, \sigma_i)$, which means that the run should be consistent with the transition function. Note that there can be different runs on the same word. Let w be an infinite sequence of elements from some set Σ, then $Inf(w) = \{\sigma | \forall i. \exists j. \text{s.t. } \sigma = w(i)\}$, where $w(j)$ is the j-th element in w. The automaton A accepts the word w iff there exists a run r of A on w s.t.

$$Inf(r) \cap F \neq \emptyset.$$

We say that $\mathcal{L}(A)$ is the set of all words accepted by A, and if $\mathcal{L}(A) = \emptyset$ then we say the language of A is empty.

Example 2.1. Consider the language $\mathcal{L}((a+b)a^\omega)$, which contains all words with finitely many b. A Büchi automaton accepting this language is shown in Figure 2.1, where, q_0 is the initial state and q_1 the accepting state. It stays for for some time in q_0 and nondeterministically picks a point in time when no b will occur anymore. If the word does not allow this, because it has infinitely many b, the automaton can never leave q_0 and rejects the word.

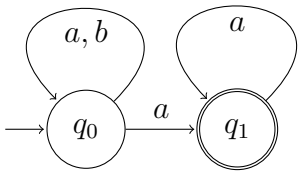

Figure 2.1: Büchi Automaton accepting $\mathcal{L}((a+b)a^\omega)$

Nondeterministic co-Büchi Automata While a Büchi automaton accepts a word if some state $q \in F$ is seen infinitely often, a co-Büchi automaton rejects if this happens. Formally a co-Büchi automaton A is a tuple $(\Sigma, Q, \delta, I, F)$, where F is the set of rejecting states. The automaton accepts a word if there is a run r on it s.t.

$$Inf(r) \cap F = \emptyset.$$

Nondeterministic Parity Automata Parity automata can bee seen as a generalization of Büchi and co-Büchi automata. A parity automaton is $A = (\Sigma, Q, \delta, I, c)$, where Q, Σ, δ, I are as before and $c : Q \to \mathbb{N}$ is a coloring function used to define which words are accepted. The parity automaton A accepts a word w iff there exists a run r s.t.

$$max\{c(q)|q \in Inf(r)\} \text{ is even.}$$

If we have a Büchi automaton $A = (\Sigma, Q, \delta, I, F)$ we can define a parity automaton $A' = (\Sigma, Q, \delta, I, c)$ s.t. $L(A) = L(A')$, where

$$c(q) = \begin{cases} 2 & \text{if } q \in F \\ 1 & \text{else.} \end{cases}$$

We can also encode co-Büchi automata as parity automata. Let $A = (Q, \Sigma, \delta, I, R)$ be a co-Büchi automaton, then we can define a parity automaton $A' = (\Sigma, Q, \delta, I, c)$ s.t. $L(A) = L(A')$, where

$$c(q) = \begin{cases} 1 & \text{if } q \in R \\ 0 & \text{else.} \end{cases}$$

Example 2.2. Consider the alphabet $\Sigma = \{a, b, c\}$ and the language $\mathcal{L}_1 = \{w|a \in Inf(w) \text{ and } b \notin Inf(w)\}$, with all words that have infinitely many a, finitely many b and any number of c. In Figure 2.2 a parity word automaton is shown that accepts the language, where the colors are given inside the states, below the state names. The automaton requires to visit q_a infinitely often, as q_a has the only even color and it forbids to visit q_b infinitely often as $c(q_b) > c(q_a)$.

Universal Automata While nondeterministic automata accept a word iff there exists an accepting run on the word, universal automata accept a word iff all runs on the word are accepting.

Deterministic Automata We call an automaton deterministic if the transition function δ always defines exactly one successor. Formally, an automaton $A = (\Sigma, Q, \delta, I, \alpha)$ is deterministic if $\forall q \in Q, \forall \sigma \in \Sigma. |\delta(q, \sigma)| = 1$ and $|I| = 1$, where α is any acceptance condition. In Figure 2.2 we see a deterministic automaton parity automaton.

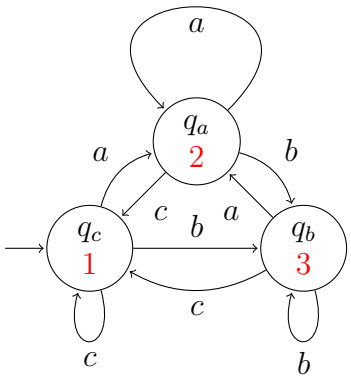

Figure 2.2: Parity Automaton accepting $\mathcal{L}_1 = \{w | a \in Inf(w) \text{ and } b \notin Inf(w)\}$

Alternating Automata

A nondeterministic automaton chooses always one arbitrary successor. An alternating automaton is restricted to pick only one successor state. It can pick several successor states, and for the run to be accepting it has to satisfy the acceptance condition from all of the successor states. For example a transition could be $\delta(q, a) = q_1 \vee q_2 \wedge q_3$, which means that the automaton will either continue from state q_1 or from both q_2 and q_3.

Let $\mathbb{B}^+(Q)$ be the set of positive (no negation) boolean formulas, with elements from Q as predicates. Note that *true* and *false* are in $\mathbb{B}^+(Q)$ as empty disjunction and empty conjunction. Formally, an alternating automaton $A = (\Sigma, Q, \delta, \{q_{init}\}, \alpha)$, where Σ, Q are as above, q_{init} is the initial state, α is the acceptance condition and $\delta : Q \times \Sigma \to \mathbb{B}^+(Q)$ is the transition function. With alternating automata we have a single initial state, because with multiple initial states it would not be clear whether they are conjunctively, or disjunctively connected. This does not affect the expressiveness.

A run $\langle T_r, r \rangle$ of A on w is a Q-labeled tree of arbitrary branching (see Section 2.1.2) satisfying the conditions

- $r(\epsilon) = q_{init}$
- $\{q_0, \ldots, q_n\} \models \delta(q, w(depth(x)))$, where $\{q_0, \ldots, q_n\}$ is the set of labels of the children of some node x in the run with $r(x) = q$ and $w(i)$ is the letter in w at the position i and $depth(x)$ is the distance of x from the root.

The conjunctions in the transition function are represented by branches in the tree, while the disjunctions are represented by different trees. A run $\langle T_r, r \rangle$ of an alternating automaton is accepting iff all the infinite branches in $\langle T_r, r \rangle$ satisfy the acceptance condition α.

Example 2.3. Let us again consider the language $\mathcal{L}_1 = \{w | a \in Inf(w) \text{ and } b \notin Inf(w)\}$. In Figure 2.3 we show an alternating Büchi automaton accepting $\mathcal{L}_1$, where from q_0 the automaton conjunctively enters q_1 and q_3. The automaton can be understood as consisting of two components. The first with the states q_1, q_2 ensures that infinitely many a are seen, and the second with the states q_3, q_4 ensures that finitely many b are seen.

Let us take a look at two example runs in seen Figure 2.4 on the word $w : aba^\omega$. The run in Figure 2.4a is accepting, because in both infinite branches of the tree, an accepting

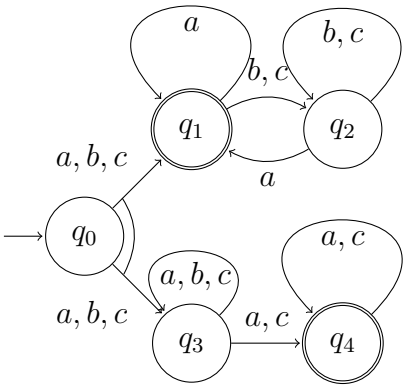

Figure 2.3: Alternating Büchi Automaton accepting $\mathcal{L}_1 = \{w | a \in Inf(w) \text{ and } b \notin Inf(w)\}$

state is seen infinitely often. The right run in Figure 2.4b is not accepting, because in the right branch the automaton never visits an accepting state.

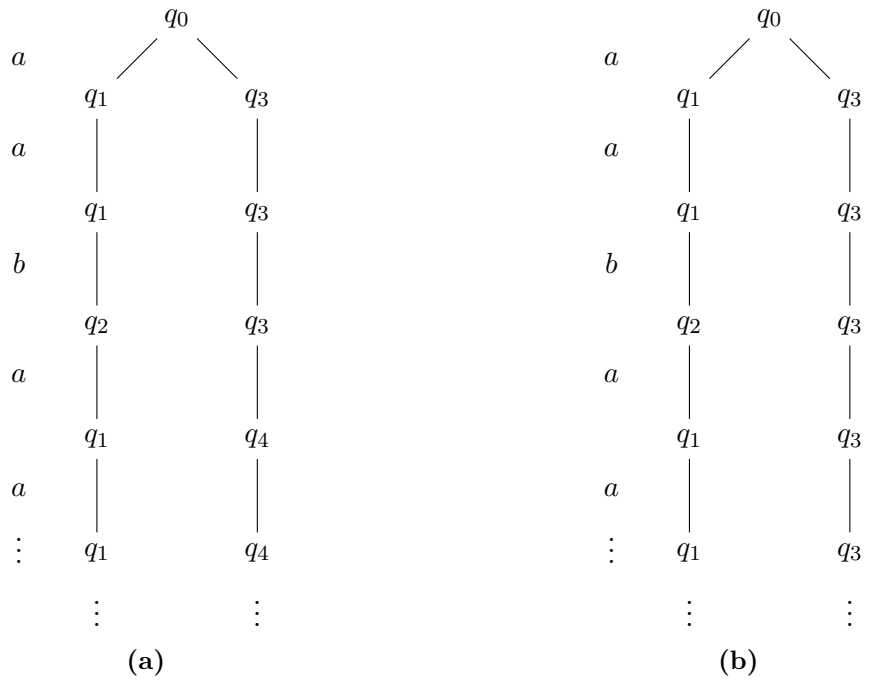

Figure 2.4: Two runs of the alternating automaton in Figure 2.3 on aba^ω. The left run is accepting, the right is not

2.1.2 Automata on Infinite Trees

Tree automata are a generalization of word automata, where a word automaton can be seen as a tree automaton on unary trees. In tree automata the automaton is able to choose a successor direction, additionally to the successor state. First we introduce nondeterministic, universal and deterministic tree automata, which we summarize as nonalternating tree automata. Then we introduce alternating tree automata. For tree automata we use

also use Büchi, co-Büchi and parity acceptance conditions, as for word automata. We only introduce automata on binary trees formally, however it is not difficult to extend the definition to trees of higher branching. We explicitly point out when we use the fact that we are working on binary trees.

Trees For now we only consider binary trees. Let $\Upsilon_2 = \{L, R\}$ be the set of directions, then a tree is a prefix closed set $T \subseteq \Upsilon_2^*$, where the root is denoted by ϵ. So a tree is a set of words, where the letters are directions and the words define the nodes in the tree. Prefix closed means $x.\upsilon \in T \implies x \in T$ with $x \in \Upsilon_2^*, \upsilon \in \Upsilon_2$. Sometimes we omit the dot (.) used for concatenation. Let Σ be a set of labels and $l : \Upsilon_2^* \to \Sigma$ a labeling function, then a labeled tree is $t = \left\langle T, l \right\rangle$ and we say t is a Σ-labeled Υ_2-tree. We call a tree full if $\exists \upsilon \in \Upsilon_2 . x\upsilon \in T \implies \forall \upsilon' \in \Upsilon_2 . x\upsilon' \in T$, so if a node has a child for some direction then it has children for all directions. When t is a full and infinite tree then $T = \Upsilon_2^*$.

Non-Alternating Tree Automata A tree automaton over full, infinite Σ-labeled Υ_2-trees is $A = (\Upsilon_2, \Sigma, Q, \delta, I, \alpha)$ where Υ_2 is the set of directions, Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q \times Q$ is the transition function (here we use the assumption of binary trees), I is the set of initial states and α is the acceptance condition. This means, the transition functions assigns state-direction tuples as successors. A run of A on the Σ-labeled Υ-tree $\left\langle \Upsilon_2^*, l \right\rangle$ is an infinite Q-labeled Υ_2-tree $\left\langle \Upsilon_2^*, r \right\rangle$, where

- $r(\epsilon) \in I$,
- $\forall x \in \Upsilon_2^* . (r(x), l(x), r(x.L), r(x.R)) \in \delta(r(x), l(x))$, which means that it should be consistent with the transition function.

A run is accepting iff all branches on the tree satisfy the acceptance condition. Like for word automata we have different branching behaviors:

Nondeterministic A tree is accepted if there exists an accepting run

Universal A tree is accepted if all runs are accepting

Deterministic If $\forall q \in Q, \forall \sigma \in \Sigma . |\delta(q, \sigma)| = 1$ and $|I| = 1$.

Alternating Tree Automata

An alternating tree automaton over full, infinite Σ-labeled Υ_2-trees is a tuple $A = (\Upsilon_2, \Sigma, Q, \delta, \{q_{init}\}, \alpha)$, where $\Upsilon_2, Q, \Sigma, \alpha$ are as before and $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon_2)$ is the transition function. An alternating tree automaton may not visit a node of the input tree at all, or it may enter a node with several conjunctive states. Thus the run of an alternating tree automaton may have a different structure than the input tree. A run $\left\langle T, r \right\rangle$ on a tree $\left\langle \Upsilon_2^*, l \right\rangle$ is a $Q \times \Upsilon_2^*$-labeled tree of arbitrary branching with the following properties: Let $\{(q_0, y\upsilon_0), \ldots, (q_n, y\upsilon_n)\}$ be the set of labels of the children of some node x in the run with $r(x) = (q, y)$. Then

- $r(\epsilon) = (q_{init}, \epsilon)$ and
- $\{(q_0, \upsilon_0), \ldots, (q_n, \upsilon_n)\} \models \delta(q, l(y))$.

A run is accepting iff all of its infinite branches satisfy the acceptance condition.

Often we work on trees that are not full. An automaton over possibly not full trees is a tuple $A = (\Upsilon_2, \Sigma, Q, \delta_\Upsilon, \ldots, \delta_\Gamma, \ldots, \delta_\emptyset, \{q_I\}, \alpha)$, where $\delta_\Gamma(q, \sigma)$ with $\Gamma \subseteq \Upsilon_2$ are the transitions used when a node x exactly has the children $x.\upsilon$ for $\upsilon \in \Gamma$. Given a possibly not full tree $\left\langle T, l \right\rangle$ with $T \subseteq \Upsilon_2^*$, $l : T \to \Sigma$ we define its full version as $\mathit{full}(\left\langle T, l \right\rangle) = \left\langle \Upsilon_2^*, l' \right\rangle$ with $l' : \Upsilon_2^* \to \Sigma \cup \{\bullet\}$, $\bullet \notin \Sigma$ and

$$l'(x) = \begin{cases} l(x) & \text{if } x \in T \\ \bullet & \text{else.} \end{cases}$$

This means, the full version of a tree, adds a padding to the original tree and labels the new nodes with $\bullet$. This makes a finite tree infinite. Now we can state the following lemma.

Lemma 2.1. *Given an automaton $A = (\Upsilon_2, \Sigma, Q, \delta_\Upsilon, \ldots, \delta_\Gamma, \ldots, \delta_\emptyset, \{q_{init}\}, c)$ over possibly not full trees we can create an alternating automaton $A_{full} = (\Upsilon_2, \Sigma \cup \{\bullet\}, Q \cup \{q_\bullet, \overline{q_\bullet}\}, \delta'', \{q''_{init}\}, c'')$ over full trees s.t. $t \in \mathcal{L}(A) \iff \mathit{full}(t) \in \mathcal{L}(A_{full})$. The state space of A_{full} is 2 states larger and the number of colors increased by at most 1.*

Proof. We use an intermediate deterministic tree automaton A', which runs on full trees where the nodes are labeled by the information in which directions a node had children before making the tree full. A' verifies that the additional information is correct. It is defined as $A' = (\Upsilon_2, \Sigma \cup \{\bullet\} \times 2^{\Upsilon_2}, \{\overline{q_\bullet}, q_\bullet\}, \delta', \{\overline{q_\bullet}\}, c')$, where the transition function is

$$\begin{aligned}
&\delta'(\overline{q_\bullet}, (\sigma, \{L, R\})) = (\overline{q_\bullet}, \overline{q_\bullet}) && \text{if } \sigma \neq \bullet \\
&\delta'(\overline{q_\bullet}, (\sigma, \{L\})) = (\overline{q_\bullet}, q_\bullet) && \text{if } \sigma \neq \bullet \\
&\delta'(\overline{q_\bullet}, (\sigma, \{R\})) = (q_\bullet, \overline{q_\bullet}) && \text{if } \sigma \neq \bullet \\
&\delta'(\overline{q_\bullet}, (\sigma, \emptyset)) = (q_\bullet, q_\bullet) && \text{if } \sigma \neq \bullet \\
&\delta'(q_\bullet, (\bullet, \emptyset)) = (q_\bullet, q_\bullet), &&
\end{aligned}$$

and all other transitions map to *false*. We use our assumption of binary trees to give a nicer definition of the transition function. The first four lines ensures that, by following the directions in the different Γ the automaton always reads labels from Σ. Together with the last line this ensures that exactly those nodes, reachable by following directions in the different Γ are labeled by Σ and all others by $\bullet$. The acceptance condition is $c'(q_\bullet) = c'(\overline{q_\bullet}) = 0$. We define a second automaton $A'' = (\Upsilon_2, \Sigma \cup \{\bullet\} \times 2^{\Upsilon_2}, Q, \delta'', \{q_{init}\}, c)$ which simulates A on the trees with additional labeling. The transition function is

$$\delta''(q, (\sigma, \Gamma)) = \begin{cases} \delta_\Gamma(q, \sigma) & \text{if } \sigma \neq \bullet \\ \mathit{false} & \text{else} \end{cases}$$

and the acceptance condition is the same as for A. A_{full} is the intersection of A' and A'', where 2^{Υ_2} is projected away from the labels. □

2.1.3 Automata on Finite Trees

For tree automata to run infinitely on finite trees we add a special direction *up*, these automata are called two-way tree automata. In this thesis we work with two-way alternating tree automata as defined by Madhusudan in [10]. Madhusudan refers to [18] to transform

his two-way alternating tree automata to nondeterministic parity tree automata with a single exponential increase of the statespace. In [4] the reader can find a good and more detailed explanation of the transformation. However, the automata from Madhusudan differ from the ones Vardi defines in [18]. First, we introduce Vardi's automata, then Madhusudan's automata and then we give a construction to transform the first into the later kind.

In this thesis we only use two-way universal tree automata and two-way alternating tree automata (as defined by Madhusudan), so we only introduce these automata types. We will define two-way universal tree automata as a special case of two-way alternating tree automata, because else the definition of two-way universal tree automata becomes complicated. Let the direction from a child up to a parent be U and let the following be its definition:

- We always assume $U \notin \Upsilon_2$,
- $\epsilon.U$ is undefined,
- $x.\upsilon.U = x$ for $\upsilon \in \Upsilon_2$.

Vardi's Two-Way Alternating Parity Tree Automata

Vardi defines a two-way alternating tree automaton as $A = (\Upsilon_2^U, \Sigma, Q, \delta, \{q_{init}\}, c)$, where $\Upsilon_2^U = \Upsilon_2 \cup \{U\}$ and the transition function is $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon_2^U)$. A run $\left\langle T_r, r \right\rangle$ on a tree $\left\langle T, l \right\rangle$ with $T \subseteq \Upsilon_2^*$ is a $Q \times T$-labeled tree of arbitrary branching with the following properties: Let x with $r(x) = (q, y)$ be a node of the run and let $\{(q_0, y\upsilon_0), \ldots, (q_n, y\upsilon_n)\}$, be the set of labels of the children of x, where $\upsilon_0, \ldots, \upsilon_n \in \Upsilon_2^U$ and $y\upsilon_0, \ldots y\upsilon_n \in T$. Then

- $r(\epsilon) = (q_{init}, \epsilon)$,
- $\{(q_0, \upsilon_0), \ldots, (q_n, \upsilon_n)\} \models \delta(q, l(y))$.

A run is accepting iff all of its infinite branches satisfy the acceptance condition.

Madhusudan's Two-Way Alternating Tree Automata

In Madhusudan's definition of two-way alternating tree automata the transition function has an additional input. We call the additional input to the transition function the *from-direction.* It defines in which direction the node lies, where the automaton was previously. Madhusudan defines a two-way alternating tree automaton as $A = (\Upsilon_2^U, \Sigma, Q, \delta_\Upsilon, \ldots, \delta_\Gamma, \ldots, \delta_\emptyset, \{q_{init}\}, c)$, where the transition functions are $\delta_\Gamma : Q \times \Sigma \times \Upsilon_2^U \to \mathbb{B}^+(Q \times \Upsilon_2^U)$. A run $\left\langle T_r, r \right\rangle$ on a tree $\left\langle T, l \right\rangle$ with $T \subseteq \Upsilon_2^*$ is a $Q \times T \times \Upsilon_2^U$-labeled tree of arbitrary branching with the following properties: Let x with $r(x) = (q, y, \upsilon)$ be a node of the run and let $\{(q_0, y\upsilon_0, \upsilon_0'), \ldots, (q_n, y\upsilon_n, \upsilon_n')\}$, be the set of labels of the children of x, where $\upsilon_0, \ldots, \upsilon_n, \upsilon_0', \ldots, \upsilon_n' \in \Upsilon^U$ and $y, y\upsilon_0, \ldots, y\upsilon_n \in T$. Then

- $r(\epsilon) = (q_{init}, \epsilon, U)$,
- $y\upsilon_i\upsilon_i' = y$ for $i \in 0, \ldots, n$, which intuitively means that υ_i' is the direction the automaton came from and
- $\{(q_0, \upsilon_0), \ldots, (q_n, \upsilon_n)\} \models \delta(q, l(y), \upsilon)$.

As before, a run is accepting iff all of its infinite branches satisfy the acceptance condition. We point out that Madhusudan's automata are a generalization of Vardi's automata. In the following we show how we can transform Madhusudan's automata to work on full trees. The construction is almost the same as in Lemma 2.1.

Lemma 2.2. *Given a two-way automaton* $A^M = (\Upsilon_2, \Sigma, Q, \delta_\Upsilon, \ldots, \delta_\Gamma, \ldots, \delta_\emptyset, \{q_{init}\}, c)$ *corresponding to Madhusudan's definition over possibly not full trees we can create a two-way alternating automaton* $A^M_{full} = (\Upsilon_2, \Sigma \cup \{\bullet\}, Q \cup \{q_\bullet, \overline{q_\bullet}\}, \delta'', \{q''_{init}\}, c'')$ *over full trees s.t.* $t \in \mathcal{L}(A^M) \iff full(t) \in \mathcal{L}(A^M_{full})$. *The state space of* A^M_{full} *is* 2 *states larger and the number of colors increased by at most* 1.

Proof. We take A' from Lemma 2.1 and we define a second automaton $A'' = (\Upsilon_2, \Sigma \cup \{\bullet\} \times 2^{\Upsilon_2}, Q, \delta'', \{q_{init}\}, c)$ which simulates A^M on the trees with additional labeling. The transition function is

$$\delta''(q, (\sigma, \Gamma), \upsilon) = \begin{cases} \delta_\Gamma(q, \sigma, \upsilon) & \text{if } \sigma \neq \bullet \\ \mathit{false} & \text{else} \end{cases}$$

and the acceptance condition is the same as for A^M. A^M_{full} is the intersection of A' and A'', where 2^{Υ_2} is projected away from the labels. □

Further, we can transform automata corresponding to Madhusudan's definition to automata corresponding to Vardi's definition.

Lemma 2.3. *Given a two-way alternating tree automaton* A^M *with additional information, as defined by Madhusudan, we can create a two-way alternating tree automaton* A^V, *which corresponds to Vardi's definition s.t.* $t \in \mathcal{L}(A^M) \iff full(t) \in \mathcal{L}(A^V)$. *The state space increases linearly in* $|\Upsilon_2|$ *and the number of colors increase by* 1.

Proof. With Lemma 2.2 and A^M we create $A^M_{full} = (\Upsilon_2^U, \Sigma \cup \{\bullet\}, Q_{full}, \delta_{full}, \{q_{init}\}, c_{full})$. For a tree $\left\langle T, l \right\rangle$ the authors define in [8] $xray(\left\langle T, l \right\rangle) = \left\langle T, l' \right\rangle$ with $l' : \Upsilon_2 \to \Sigma \times \Upsilon$ as

$$\begin{aligned} l'(\epsilon) &= (l(\epsilon), \upsilon_\epsilon) \\ l'(x.\upsilon) &= (l(x.\upsilon), \upsilon), \end{aligned}$$

where $\upsilon_\epsilon \in \Upsilon_2$ is some default direction. So *xray* adds information about the direction of a node to the nodes label. We define the one-way deterministic tree automaton $A_{xray} = (\Upsilon_2, (\Sigma \cup \{\bullet\}) \times \Upsilon_2, \Upsilon_2, \delta_{xray}, \{\upsilon_\epsilon\}, c_{xray})$, which accepts the *xray* of $\Sigma \cup \{\bullet\}$-labeled Υ_2-trees. The transition function is

$$\delta_{xray}(\upsilon, (\sigma, \upsilon')) = \begin{cases} (L, R) & \text{if } \upsilon = \upsilon' \\ \mathit{false} & \text{else.} \end{cases}$$

Note that we use our assumption of binary trees to write the transition function nicer. The automaton goes to all nodes, and ensures that the last direction the automaton took to reach the node is the same as the direction in the nodes label. The acceptance condition is $c_{xray}(L) = c_{xray}(R) = 0$. We define the two-way alternating automaton $A^V_1 = (\Upsilon_2^U, (\Sigma \cup \{\bullet\}) \times \Upsilon_2, Q_{full} \times \Upsilon_2^U, \delta', \{(q_{init}, U)\}, c')$ corresponding to Vardi's definition,

which remembers in its state space the last direction the automaton went. The automaton uses this information to simulate A^M. The transition function is

$$\delta' : (Q_{full} \times \Upsilon_2^U) \times (\Sigma \cup \{\bullet\} \times \Upsilon_2) \to \mathbb{B}^+(Q_{full} \times \Upsilon_2^U)$$
$$\delta'((q, \upsilon'), (\sigma, \upsilon'')) = \{((q_0, \upsilon_0), \upsilon_0), \ldots, ((q_{n-1}, \upsilon_{n-1}), \upsilon_{n-1})\},$$

where $\delta_{full}(q, \sigma, \upsilon''') = \{(q_0, \upsilon_0), \ldots, (q_{n-1}, \upsilon_{n-1})\}$ and $\upsilon''' = \begin{cases} U & \text{if } \upsilon' \in \Upsilon_2 \\ \upsilon'' & \text{else.} \end{cases}$ If the automaton went down in the last transition, then $\upsilon' \in \Upsilon$ and we give Madhusudan's transition function the additional information that the automaton came from above. If $\upsilon' = U$ we give the information from which direction we came up and that is the current nodes direction, as defined by *xray*. The acceptance condition is $c'((q, \upsilon)) = c_{full}(q)$, which means that it ignores the directions in the labels. We take the intersection of A_1^V and A_{xray} and project the directions in the labels away and get A^V. □

Lemma 2.4 ([10])**.** *Given a two-way tree automaton (corresponding to Madhusudan's definition of two-way automata) A^M on finite, possibly not full trees we can create a one-way tree automaton A' s.t. $\mathcal{L}(A^M) = \mathcal{L}(A')$, where A' is exponentially larger than A^M.*

Proof.
1. From A^M create A^V using Lemma 2.3. A^V corresponds to Vardi's definition of two-way automata and accepts trees with an infinite padding of $\bullet$-labeled nodes.
2. From A^V we create $A = (\Upsilon_2, \Sigma, Q, \delta, I, c)$ using Vardi's transformation of two-way automata to one-way automata [18]. A is a one-way parity tree automaton accepting trees with an infinite padding of $\bullet$-labeled nodes.
3. Identify the set of states $Q_\bullet \subseteq Q$ from where the full, infinite $\{\bullet\}$-labeled Υ_2-tree is accepted. Then the finitely running tree automaton $A' = (\Upsilon_2, \Sigma, Q, \delta, I, Q_\bullet)$ accepts the Σ-labeled Υ_2-tree t iff $full(t) \in \mathcal{L}(A)$.

□

Universal Two-Way Tree Automata A two-way universal tree automaton is a two-way alternating tree automaton $A = (\Upsilon_2^U, \Sigma, Q, \delta_\Upsilon, \ldots, \delta_\Gamma, \ldots, \delta_\emptyset, I, \alpha)$, where $\delta_\Gamma : Q \times \Sigma \times \Upsilon_2^U \to \mathbb{B}^+(Q \times \Upsilon_2^U)$ uses only conjunctions, but no disjunctions.

2.2 Linear Temporal Logic

We briefly introduce linear temporal logic (LTL) [12]. Let AP be a set of propositional variables and let $p \in AP$. Then a LTL formula is defined as

$$\langle\varphi\rangle ::= p \mid \langle\varphi\rangle \vee \langle\varphi\rangle \mid \neg\langle\varphi\rangle \mid \bigcirc\langle\varphi\rangle \mid \langle\varphi\rangle \,\mathcal{U}\, \langle\varphi\rangle.$$

Further we define the usual abbreviations

$$\begin{aligned} \mathit{true} &\equiv \quad p \vee \neg p, \\ \mathit{false} &\equiv \qquad \neg p, \\ \Diamond\varphi &\equiv \mathit{true}\,\mathcal{U}\,\varphi, \\ \Box\varphi &\equiv \quad \neg\Diamond\neg\varphi. \end{aligned}$$

The semantics is defined over words $w \in \Sigma^\omega$ with $\Sigma = 2^{AP}$. Let $w[j\ldots] = w_j w_{j+1}\ldots$ be the suffix of the word w starting at the index j, then we define the semantics as

$$\begin{aligned} w &\models p && \text{if} \quad p \in w(0), \\ w &\models \neg\varphi && \text{if} \quad w \not\models \varphi, \\ w &\models \varphi \vee \varphi' && \text{if} \quad w \models \varphi \text{ or } w \models \varphi', \\ w &\models \bigcirc\varphi && \text{if} \quad w[1\ldots] \models \varphi, \\ w &\models \varphi\,\mathcal{U}\,\varphi' && \text{if} \quad \exists i.\forall j.0 \leq j < i \implies w[j\ldots] \models \varphi \text{ and } w[i\ldots] \models \varphi'. \end{aligned}$$

We point out that LTL formulae can be transformed into a Büchi automaton s.t. the set of words satisfying the LTL formula is equal to the set of words accepted by the Büchi automaton (see e.g. [7]).

2.3 Distributed Synthesis

In this section we briefly explain the distributed synthesis of reactive programs. Further we also introduce the proof of undecidability by Pnueli and Rosner of the general distributed synthesis. Later in this thesis we will adapt this undecidability proof for programs.

First we provide some definitions. An architecture is a tuple $\mathcal{A} = (P, P_0, E, O)$, where P is a set of processes, $P_0 \in P$ is the environment process, $E \subseteq P \times O \times P$ is a set of directed edges and O is a set of variables. We assume that every variable labels exactly one edge. We assume that communication involves a delay of one discrete step. This means the values process P_j reads in step $i+1$ have been produced by other processes in step i. Further, in step 0 all processes read default values. For processes $P_1, \ldots, P_n$ and the environment P_0 the specification automaton A_φ is defined over the alphabet $\{0,1\}^{|O|}$. For sets I, O a strategy is a function $s : I^* \to O$ that assigns a sequence of values an output. Let process P_j produce values for process P_k and let s_j, s_k be strategies for the processes and let i be a sequence of inputs. Then we get the computation

$$\begin{pmatrix} s_j(\epsilon) \\ s_k(\epsilon) \end{pmatrix} \begin{pmatrix} s_j(i_0) \\ s_k(s_j(\epsilon)) \end{pmatrix} \begin{pmatrix} s_j(i_0 i_1) \\ s_k(s_j(\epsilon) s_j(i_0)) \end{pmatrix} \ldots.$$

A class of architectures we often use are the hierarchical architectures, which are defined as all architectures where P_{j+1} only receives values from P_j and the environment may receive values from any process.

2.3.1 Undecidability of Distributed Synthesis of Transition Systems

Here we introduce the proof that the synthesis of distributed systems in general is undecidable. The proof was first developed by Pnueli and Rosner in [13] and later reformulated

by Finkbeiner and Schewe in [6]. Here we only state the proof introduced in [6]. The general idea is, for a given Turing machine M we synthesize a transition system that simulates M. Further the specification is defined s.t. the problem is realizable iff the Turing machine eventually halts. Thus, the halting problem has been reduced to the problem of distributed synthesis.

Assume we are given a deterministic Turing machine M and have the architecture

$$\mathcal{A}_0 = (\{P_0, P_1, P_2\}, P_0, \{(P_0, x_1, P_1), (P_1, y_1, P_0), (P_0, x_2, P_2), (P_2, y_2, P_0)\}, \{x_1, x_2, y_1, y_2\}),$$

where the environment process is not drawn and edges without head or tail originate or end in the environment process (see Figure 2.5). Let the domain for the input values be

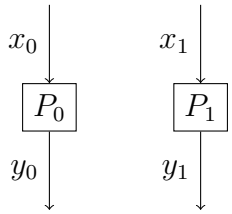

Figure 2.5: Architecture $\mathcal{A}_0$

$\{start, \overline{start}\}$, where *start* means that the the program should *start* to output a configuration for M (consisting of tape content, head position and state) and further *start*-inputs mean nothing. The value $\overline{start}$ indicates that the system should not yet start to output configurations. The domain of the output is $Q_M \cup \Gamma_M$, where Q_M is the set of states, Γ_M is the set of tape symbols of M with $\bullet$ being the blank-symbol. Assume M is in state q and the tape has the content as shown below

$$\cdots \bullet \bullet \gamma_0 \gamma_1 \ldots \gamma_{i-1} \overset{\downarrow}{\gamma_i} \ldots \gamma_{n-1} \bullet \bullet \ldots$$

and the head is above the symbol γ_i. Then the configuration is output by first producing the symbols $\gamma_0, \ldots, \gamma_{i-1}$ left of the head, then the state q and then the symbols $\gamma_i, \ldots, \gamma_{n-1}$ and the first $\bullet$ left of γ_{n-1}.

Let C_i^1 be the ith configuration produced by P_1 and similarly C_j^2 for P_2 and let $\perp$ be the accepting state of M. Let ψ_1 be the specification that demands that

- P_1 initially outputs $\perp$ until the first S is received.
  - When the first *start*-signal is received P_1 outputs the initial configuration of M and then P_1 outputs the second configuration of M (as defined by the transition table).
  - After that, if P_1 outputs a terminating configuration (which contains $\perp$) it will always output this configuration.
- If $C_i^1 \vdash C_j^2$ holds and are started to be output at the same time, then also $C_{i+1}^1 \vdash C_{j+1}^2$ holds

and let ψ_2 be the specification that demands the same; only that P_2 takes the role of P_1 and C_1^i, C_2^j are swapped and C_{i+1}^1, C_{j+1}^2 are swapped as well.

All transition systems which realize the specification $\psi_1 \wedge \psi_2$, output the computation of M on the empty input. Finkbeiner and Schewe prove this by induction on the number of configurations produced according to the computation of M. For $n = 0$ the first configuration produced corresponds to the first configuration of the computation because of the specifications ψ_1 and ψ_2. Let $n' = n + 1$, where we know by induction hypothesis that the first n configurations are produced correctly. Let the first $n + 1$ configurations produced be

$$C_0^1 C_1^1 \dots C_{n-1}^1 C_n^1$$

with $C_i^1 \vdash C_{i+1}^1$ for $i \in \{0, \dots, n-1\}$. Now the environment sends the *start*-signal to P_2 s.t. C_n^1 and C_{n-1}^2 are started to be output at the same time. We get the series of outputs

$$\begin{array}{cccccc} C_0^1 & C_1^1 & C_2^1 & & C_{n-1}^1 & C_n^1 \\ \bot & C_0^2 & C_1^2 & \cdots & C_{n-2}^2 & C_{n-1}^2 \end{array},$$

and by induction hypothesis we have $C_{n-1}^2 \vdash C_n^1$. Now ψ_2 ensures that $C_n^2 \vdash C_{n+1}^1$ holds, which completes the induction. The proof uses the fact that the processes do not have any information of the other processes input and that the specification has to be realized for every possible behavior of the environment.

We define $\psi_0 \wedge \psi_1 \wedge \psi_2$, where ψ_0 demands that P_1 and P_2 output $\bot$ infinitely often. $\psi_0 \wedge \psi_1 \wedge \psi_2$ is realizable over $\mathcal{A}_0$ iff M halts, because if M does not halt $\bot$ will not be produced infinitely often as the systems produce the computation of M.

2.4 Synthesis of Reactive Programs

Here we introduce the synthesis of reactive programs as defined by Madhusudan in [10]. The general idea is to run ω-tree automata on syntax trees of finite programs to infinitely interpret programs, where the program memory is stored in the statespace of the tree automaton. To be able to run infinitely on a finite tree the automaton has to be able to go up a tree. For the emptiness check these two-way alternating tree automata are translated into nondeterministic tree automata with exponential blowup. We point out that we often use programs and syntax trees interchangeably.

Madhusudan defines three automata for the synthesis:

A_{sim} is a two-way alternating Büchi tree automaton. It infinitely interprets the program with all possible inputs. At the same time simulates $A_{\neg\varphi}$ with the inputs and outputs the program receives and sends during interpretation. It accepts programs violating $A_{\neg\varphi}$ for some input sequence.

$A_{non-reac}$ also is a two-way alternating Büchi tree automaton. It infinitely interprets the program with all possible inputs and accepts non-reactive programs . A program is non-reactive if the inputs and outputs do not always alternate or if it terminates (see Section 2.4.3).

A_{pgm} is a one-way deterministic tree automaton of any acceptance condition. It accepts syntax trees that represent programs.

Then

$$\overline{A_{sim} \cup A_{non-reac}} \cap A_{pgm}$$

accepts syntax trees of programs realizing A_φ. The automaton $A_{sim} \cup A_{non-reac}$ accepts all syntax trees we are not interested in, so we complement it. Then $\overline{A_{sim} \cup A_{non-reac}}$ also accepts trees, which are not syntax trees so we restrict the trees to syntax trees by intersecting with A_{pgm}.

2.4.1 Programs

The specification is given by a nondeterministic Büchi word automaton A_φ over the alphabet $\{0,1\}^{N_I+N_O}$, where N_I, N_O are the number of inputs and outputs of the program we want to synthesize. Let us fix the set of variables we want to use with B, where we assume that $|B| \geq max\{N_I, N_O\}$ and that the variables are boolean. We are interested in reactive programs, which interact infinitely with the environment. For our programs we fix the syntax seen in Figure 2.6, where $b \in B$ and $\vec{b}$ is a vector of variables from B

$$
\begin{aligned}
\langle root \rangle ::= \quad & \textbf{root}\ \langle stmt \rangle \\
\langle stmt \rangle ::= \quad & \langle stmt \rangle; \langle stmt \rangle \mid \textbf{skip} \mid :=_b \langle expr \rangle \mid \textbf{input}\ \vec{b} \mid \textbf{output}\ \vec{b} \mid \\
& \quad \textbf{if}\ \langle expr \rangle\ \textbf{then}\ \langle stmt \rangle\ \textbf{else}\ \langle stmt \rangle \mid \textbf{while}\ \langle expr \rangle\ \langle stmt \rangle \\
\langle expr \rangle ::= \quad & b \mid \textbf{tt} \mid \textbf{ff} \mid \langle expr \vee expr \rangle \mid \neg \langle expr \rangle
\end{aligned}
$$

Figure 2.6: EBNF for programs

and the length of $\vec{b}$ is equal to N_I or N_O respectively. We fix the semantics formally later. The intended semantics is that a program receives values from the environment through **input** $\vec{b}$ statements, and updates the values of variables in $\vec{b}$ with values as defined by the environment. Then the program internally computes the values it wants to output, stores them in its variables and sends the values to the environment with **output** $\vec{b}$. The internal computation is finite, but may be arbitrary long.

Example 2.4. Consider the specification $\varphi : \square(i \neq o)$ over the alphabet $\{\binom{0}{0}, \binom{1}{0}, \binom{0}{1}, \binom{1}{1}\}$ with $N_I = N_O = 1$. The language of φ is $\mathcal{L}(\varphi) = \left(\binom{1}{0} + \binom{0}{1}\right)^\omega$, i.e. the words where the output always is different from the input. In Figure 2.7 a program is shown. It first reads

```
while tt
    input b
    b := ¬b
    output b
```

Figure 2.7: Example program

an input, negates it and then outputs it. For the input sequence $w_i : 101010\ldots$ the program produces the output sequence $w_o : 010101\ldots$. To see if the program realizes φ we have to check if for all possible input sequences the program creates an output sequence which together satisfy φ.

We want to reason about programs with tree automata, so we fix the set of labels for our trees as

$$\Sigma = \{\mathbf{root}, ;, \mathbf{if}, \mathbf{then}, \mathbf{while}, \neg, \vee, \mathbf{tt}, \mathbf{ff}\} \cup V \cup \{:=_b \mid b \in B\}$$
$$\cup \{\mathbf{input}\ \vec{b} \mid \vec{b} \in B^{N_I}\} \cup \{\mathbf{output}\ \vec{b} \mid \vec{b} \in B^{N_O}\}.$$

Let A_{pgm} accept the set of all (finite) Σ-labeled trees with correct syntax as defined by the EBNF in Figure 2.6. In Figure 2.8a we see a Σ-labeled tree, however the syntax is wrong. The if-statement should have an expression and a then-node attached. In Figure 2.8b the tree representation of the program in Figure 2.7 is shown.

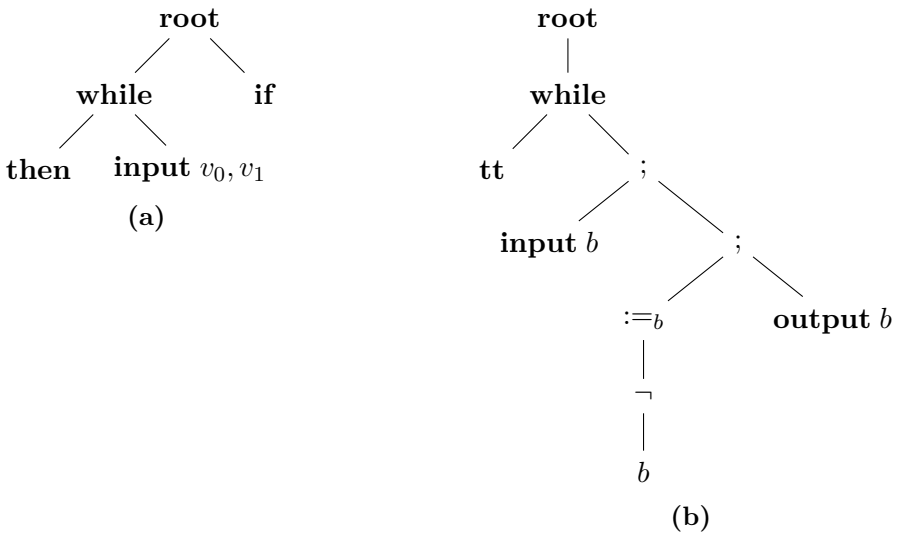

Figure 2.8: Left is a Σ-labeled tree, which is not a program tree and right is a program tree

2.4.2 Simulator

The simulator interprets a program with all possible inputs from the environment, computes the outputs and simultaneously checks whether there exist inputs with which the property is violated. The tree automaton accepts all programs violating the property for some inputs. Later the automaton is negated.

Let $A_{\neg\varphi} = (\Sigma, Q_{\neg\varphi}, \{q_{init}\}, \delta_{\neg\varphi}, F_{\neg\varphi})$ be the nondeterministic Büchi word automaton that accepts the negation of the property φ. Let B be the set of variables the synthesized program may use and let N_I, N_O be the input and output arities. Further let $s : B \to \{0, 1\}$ be a valuation for the program variables, let S be the set of all possible valuations and let $I = 2^{\{0,1\}^{N_I}}$ be the set of all possible inputs a program can read. We point out that Madhusudan defined $S = \{0, 1\}^{|B|}$.

The statespace consists of the computation part ($S \times \{0, 1\}$), which is used to evaluate the values of expressions and of a control part ($S \times Q_{\neg\varphi} \times I \times \{inp, out\} \times \{fin, \overline{fin}\}$), where $\{inp, out\}$ is used to remember whether the next I/O statement should be an input or an output and $\{fin, \overline{fin}\}$ determines together with $Q_{\neg\varphi}$ whether the current state is a final state. Then the statespace of the simulator A_{sim} is

$$P : (S \times \{0, 1\}) \cup (S \times Q_{\neg\varphi} \times I \times \{inp, out\} \times \{fin, \overline{fin}\}).$$

Intuitively, the automaton walks over the program tree, interpreting the statements. At assignments the automaton updates the memory of the program, at while-loops it executes the loop and at the end of the loop goes up the tree to the beginning of the loop and executes the statements again. At an input statement the automaton updates the memory with the input values read and at an output statement it advances $A_{\neg\varphi}$ one step.

Let s be a valuation, b, b' variables and v a value then

$$s[b/v](b') = \begin{cases} v & \text{if } b' = b \\ s(b') & \text{else,} \end{cases}$$

and it is clear how this extends to vectors of variables and values. We use this to change update variable valuations. The simulator is $A_{sim} = (\Upsilon_2^U, \Sigma, P, \{p_{init}\}, \delta_{LR}, \delta_L, \delta_\emptyset, F)$, where

- $p_{init} = (s_{init}, q_{init}, i_{init}, \overline{fin})$ is the initial state, where $s_{init} \in S$ assigns to all variables 0, q_{init} is the initial state of $A_{\neg\varphi}$, $i_{init} = 0^{N_I}$ is the vector with 0 at all positions,
- $F = (S \times F_{\neg\varphi} \times I \times \{inp, out\} \times \{fin\})$ is the set of final states, meaning that a state is final if its $A_{\neg\varphi}$-part is final and the *fin*-flag is set and
- $\delta_{LR/L/\emptyset} : P \times \Sigma \times \Upsilon_2^U \to \mathbb{B}^+(P \times \Upsilon_2^U)$ are the transition functions defined in the following (we do not use δ_R).

Root We introduce the transition function δ in parts. The first transition taken is

$$\delta_L(p_{init}, \mathbf{root}, U) = (p_{init}, L)$$

and if the automaton comes from a child back to the root-node the program terminated. In this case the automaton accepts.

$$\delta_L((s, q, i, m, t), \mathbf{root}, L) = true$$

Boolean Expressions When the automaton encounters an $\vee$-expression, the automaton either tries to evaluate one of the children to *true* in case it guessed the expression is evaluates to true, or it tries to evaluate both children to *false* in case it guessed the expression is evaluates to false.

$$\delta_{LR}((s,1), \vee, U) = ((s,1), L) \vee ((s,1), R) \qquad \delta_\emptyset((s,0), \mathbf{tt}, U) = false$$
$$\delta_{LR}((s,0), \vee, U) = ((s,0), L) \wedge ((s,0), R) \qquad \delta_\emptyset((s,0), \mathbf{ff}, U) = true$$

At a $\neg$-node the automaton inverts its guess.

$$\delta_L((s,v), \neg, U) = ((s, 1-v), L)$$

When the automaton encounters the leaf of an expression it compares the value of the leaf with the value it guessed for the expression upon entering. If they are the same it accepts, else it rejects.

$$\begin{aligned} \delta_\emptyset((s,1), \mathbf{tt}, U) &= true & \delta_\emptyset((s,0), \mathbf{tt}, U) &= false \\ \delta_\emptyset((s,1), \mathbf{ff}, U) &= false & \delta_\emptyset((s,0), \mathbf{ff}, U) &= true \\ \delta_\emptyset((s,v), b, U) &= true & &\text{if } s(b) = 1 \\ &= false & &\text{else} \end{aligned}$$

I/O Statements At an input statement the automaton disjunctively reads all possible input values and remembers the values read.

$$\delta_\emptyset((s,q,i,inp,f),\mathbf{input}\,\vec{b},U) = \bigvee_{val\ over\ \vec{b}} ((s[\vec{b}/val],q,val,out,\overline{fin}),U)$$

At an output statement the automaton advances $A_{\neg\varphi}$ one step with the input it read at the last input statement and the output the automaton computed. Also, the automaton sets the *fin*-flag, which is set to $\overline{fin}$ in the next transition.

$$\delta_\emptyset((s,q,i,out,f),\mathbf{output}\,\vec{b},U) = \bigvee_{q'\in\delta_{\neg\varphi}(q,i,s(\vec{b}))} ((s,q',i,inp,fin),U)$$

Non I/O statements At a skip-statement the automaton goes to the next statement, without changing the state space

$$\delta_\emptyset((s,q,i,m,t),\mathbf{skip},U) = ((s,q,i,m,t),U)$$

At a $:=_b$-statement guesses the value of the assignment, updates the variable b and conjunctively validates the guess and continues interpretation.

$$\begin{aligned}&\delta_L((s,q,i,m,t),:=_b,U)\\ &\quad= ((s,1),L)\wedge((s[b/1],q,i,m,t),U)\vee((s,0),L)\wedge((s[b/0],q,i,m,t),U)\end{aligned}$$

When the automaton encounters an if-statement it tries to evaluate the condition to 1 and interprets the then-branch. Also, the automaton attempts to evaluate the condition to 0 and interprets the else-branch. Note that for the directions RL and RR we assume the automaton has an additional bit in its state space that stores where the automaton should go after going right. We did not add this bit to the state space, to keep it easier to understand.

$$\begin{aligned}&\delta_{LR}((s,q,i,m,t),\mathbf{if},U)\\ &\quad= ((s,1),L)\wedge((s,q,i,m,t),RL)\vee((s,0),L)\wedge((s,q,i,m,t),RR)\end{aligned}$$

At a while-loop the automaton again assumes a value of the condition and conjunctively validates the assumption. The automaton assumes the condition is true and thus tries to evaluate it to 1 and enters the loop. At the same time the automaton assumes the condition is false, tries to evaluate it to 0 and does not enter the loop. This is the same when the automaton first encounters the loop when it comes from a parent, or when it comes from the loop and checks if it should be executed again.

$$\begin{aligned}&\delta_{LR}((s,q,i,m,t),\mathbf{while},U)\\ &=\delta_{LR}((s,q,i,m,t),\mathbf{while},R)\\ &\quad= ((s,1),L)\wedge((s,q,i,m,t),R)\vee((s,0),L)\wedge((s,q,i,m,t),U)\end{aligned}$$

The sequential operator ; defines the order in which the statements should be interpreted. First the left branch is interpreted and then the right branch.

$$\begin{aligned}\delta_{LR}((s,q,i,m,t),;,U) &= ((s,q,i,m,t),L)\\ \delta_{LR}((s,q,i,m,t),;,L) &= ((s,q,i,m,t),R)\\ \delta_{LR}((s,q,i,m,t),;,R) &= ((s,q,i,m,t),U)\end{aligned}$$

When **if** and **then**-statements are encountered and the automaton comes from a child, this means the statements in the branch were completely interpreted. In this case the automaton goes up to find the next statement to be interpreted.

$$\begin{aligned}&\delta_{LR}((s,q,i,m,t),\mathbf{if},R)\\&=\delta_{LR}((s,q,i,m,t),\mathbf{then},L)\\&=\delta_{LR}((s,q,i,m,t),\mathbf{then},R)=((s,q,i,m,t),U)\end{aligned}$$

2.4.3 Non-Reac Checker

Madhusudan did not formally define $A_{non\text{-}reac}$, which checks whether a program is non-reactive. As non-reactive behavior we define

- if the first I/O statement encountered during interpretation is not an input,
- if the I/O statements do not always alternate,
- if the program terminates or
- if eventually no I/O statement encountered anymore.

Here we provide a possible definition of $A_{non\text{-}reac}$. The automaton interprets the program, just like the simulator, however it does not simulate $A_{\neg\varphi}$. Instead the automaton guesses at the start and at every I/O statement that it will not encounter another I/O statement. If any of these guesses is correct the automaton accepts. Further, $A_{non\text{-}reac}$ remembers the next I/O statement a reactive program would encounter. While A_{sim} rejects if the expectation is violated$A_{non\text{-}reac}$ accepts in this case.

The state space is

$$P : (S \times \{0,1\}) \cup (S \times \{inp, out, \overline{io}\}),$$

where everything $\overline{io}$ is used to remember that the automaton does not want to encounter another I/O statement. Then $A_{non\text{-}reac} = (\Upsilon_2^U, \Sigma, P, \{(s_{init}, inp)\}, \delta_{LR}, \delta_L, \delta_\emptyset, S \times \{\overline{io}\})$. We only give the transitions different from the ones in A_{sim}.

Root At the start a reactive program expects to see an input. Thus $A_{non\text{-}reac}$ accepts if the first transition is not an input. And it accepts if it never sees an input.

$$\delta_L(p_{init}, \mathbf{root}, U) = ((s_{init}, inp), L) \vee ((s_{init}, \overline{io}), L)$$

Also the automaton accepts if the program terminates.

$$\delta_L((s,m), \mathbf{root}, L) = true$$

I/O Statements If the automaton encounters an input when a reactive program would encounter an output the automaton accepts. Similarly when it encounters an output.

$$\begin{aligned}\delta_\emptyset((s, out), \mathbf{input}\,\vec{b}, U) &= true\\ \delta_\emptyset((s, inp), \mathbf{output}\,\vec{b}, U) &= true\end{aligned}$$

At an input statement the automaton disjunctively reads all possible input values, however $A_{non\text{-}reac}$ does not need to remember the inputs, as it does not simulate $A_{\neg\varphi}$. Additionally,

the automaton tries to validate for every guessed input that it will not encounter another I/O statement.

$$\delta_\emptyset((s, inp), \mathbf{input}\,\vec{b}, U) = \bigvee_{val\ over\ \vec{b}} \Big(((s[\vec{b}/val], out), U) \vee ((s[\vec{b}/val], \overline{io}), U)\Big)$$

At an output statement the automaton continues interpretation and disjunctively tries to validate that it will not encounter another I/O statement.

$$\delta_\emptyset((s, out), \mathbf{output}\,\vec{b}, U) = ((s, inp), U) \vee ((s, \overline{io}), U)$$

All other transitions map to *false*.

3 Synthesis of Distributed Reactive Programs

The synthesis of distributed reactive systems was first partly solved by Pnueli and Rosner in [13]. They give a beautiful proof of undecidability for the general distributed synthesis, and they solve the distributed synthesis for hierarchical architectures. In [6] the authors reformulate and extend the proof from Pnueli and Rosner. In this chapter we will adapt the proof of undecidability for distributed reactive systems from [6] to a proof of undecidability for distributed reactive programs. Also we conjecture why the decidability results from distributed synthesis of transition systems do not hold for distributed programs. Further we define a restricted programming language which is based on the programming language for which Madhusudan gave in [10] a synthesis procedure. For this restricted language we solve the synthesis of distributed programs for any architecture.

3.1 Distributed Programs

In Section 2.4.1 we define sequential programs. The programs are restricted in the number of variables to keep the alphabet and the statespace of the interpreting tree automaton finite. First we adapt our definition of architectures. Let an architecture for the distributed synthesis of reactive programs be $\mathcal{A} = (P, P_0, E, O, \{B_1, \ldots, B_n\})$, where $P = \{P_0, P_1, \ldots, P_n\}$ is a set of processes, P_0 is the environment process, $E \subseteq P \times O \times P$ is a set of labeled, directed edges, O is a set of output signals and B_j is the set of boolean variables a process P_j can use, where all B_j are disjoint. Note that we speak of output signals, instead of output variables (as in classical distributed synthesis) to avoid confusion with the variables a process may use for internal computation. Further we point out that by only considering boolean variables we do not lose expressiveness. A visualization is shown in Figure 3.2. Assume we are given an architecture $\mathcal{A} = (P, P_0, E, O, \{B_1, \ldots, B_n\})$ then

- $N_O^{j,k} = |\{o \in O | (j, o, k) \in E\}|$ is the number of output values a process P_j produces for process p_k (we assume $N_O^{j,j} = 0$),
- $N_O^j = \sum_{k=0}^{n} N_O^{j,k}$ is the number of output values a process P_j produces,
- $N_I^j = \sum_{k=0}^{n} N_O^{k,j}$ is the number of input values a processP_j reads and
- the specification is given as a nondeterministic Büchi automaton A_φ over the alphabet $\{0,1\}^{|O|}$.

For the processes in P we define reactive programs $p_1, \ldots, p_n$, where program p_j is a concrete implementation of process P_j.

$$
\begin{aligned}
\langle root \rangle &::= \quad \textbf{root}\, \langle stmt_1 \rangle \langle stmt_2 \rangle \dots \langle stmt_n \rangle \\
\langle stmt_j \rangle &::= \quad \langle stmt_j \rangle; \langle stmt_j \rangle \mid \textbf{skip} \mid b^j := \langle expr_j \rangle \mid \textbf{input}\, \vec{b}^j \mid \textbf{output}\, \vec{b}^j \mid \\
&\qquad \textbf{if}\, \langle expr_j \rangle\, \textbf{then}\, \langle stmt_j \rangle\, \textbf{else}\, \langle stmt_j \rangle \mid \textbf{while}\, \langle expr_j \rangle\, \langle stmt_j \rangle \\
\langle expr_j \rangle &::= \quad b^j \mid \textbf{tt} \mid \textbf{ff} \mid \langle expr_j \vee expr_j \rangle \mid \neg \langle expr_j \rangle
\end{aligned}
$$

Figure 3.1: EBNF for distributed programs

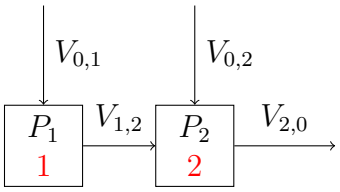

Figure 3.2: Example architecture where P_1, P_2 have one and two variables respectively

Syntax With these definitions we define the terms program p_j can use as

$$
\begin{aligned}
\Sigma_j = &\{;, \textbf{if}, \textbf{then}, \textbf{while}, \neg, \vee, \textbf{tt}, \textbf{ff}\} \cup B_j \cup \{:=_b \mid b \in B_j\} \\
&\cup \{\textbf{input}\, \vec{b} \mid \vec{b} \in B_j^{N_I^j}\} \cup \{\textbf{output}\, \vec{b} \mid \vec{b} \in B_j^{N_O^j}\}.
\end{aligned}
$$

This gives the EBNF depicted in Figure 3.1, where **root** connects the different programs. The EBNF can be recognized by a deterministic tree automaton which we call A^1_{pgm}. Note that A^1_{pgm} only accepts finite trees. Further, we can define A^1_{pgm} to be deterministic with any acceptance condition without increase of the statespace.

Semantics We assume that the communication in the distributed system is synchronized, which means that if one program arrives at an output statement, it can only continue when all programs that receive input from this program are at an input statement. We only fix the semantics for I/O statements. The other program terms have the natural semantics. Let $signals_{out}(j)$ be an arbitrarily ordered vector of the signals in $\bigcup_{k \in \{0,\dots,n\}} \{o \in O | (j, o, k) \in E\}$ and similarly $signals_{in}(j)$ is an ordered vector of the signals in $\bigcup_{k \in \{0,\dots,n\}} \{o \in O | (k, o, j) \in E\}$. Then for all programs p_j and for a statement **output** $\vec{b}$ with $\vec{b} \in \{0,1\}^{N_O^j}$ to every index $i \in N_O^j$ exactly one signal from $signals_{out}(j)$ is assigned and similarly for input statements. This defines which values are read and written by which programs and assumes that the environment also has input and output statements.

Example 3.1. Assume we are given the specification

$$\square(\bigcirc\bigcirc V_{2,0} \iff (V_{0,1}\, XOR\, \bigcirc V_{0,2}))$$

and the architecture in Figure 3.2. The specification demands that the output to the environment is equal to the XOR of input to P_1 from two steps ago and the input to P_2 from last step. Let us assume that the first few inputs from the environment are

$$
\begin{matrix} V_{0,1} \\ V_{0,2} \end{matrix} : \begin{matrix} 0101 \\ 1110 \end{matrix} \cdots
$$

```
while tt
    input b
    output b
```

```
while tt
    input b_1, b_2
    b_1 := ¬(b_1 ∨ ¬b_2) ∨ ¬(¬b_1 ∨ b_2)
    output b_1
```

Figure 3.3: A distributed program. The left program p_1 runs on P_1 and the right program p_2 runs on P_2

In Figure 3.3 we see two programs realizing the specification over the architecture (note that $v_1 \iff v_2 \equiv \neg(v_1 \vee \neg v_2) \vee \neg(\neg v_1 \vee v_2)$). The program p_1 sends on $V_{1,2}$ the values

$$V_{1,2} : 0010\ldots,$$

where $V_{1,2}[i+1] \iff V_{0,1}[i]$. The program p_2 sends the following values to the environment:

$$V_{2,0} : 0110\ldots,$$

where $V_{2,0}[i+1] \iff (V_{1,2}[i]\ XOR\ V_{0,2})$.

Later, we represent distributed programs as syntax trees to work on them with tree automata. In Figure 3.4 we show the syntax tree for the programs from Figure 3.3. The left subtree of the root represents p_1 and the right subtree represents p_2.

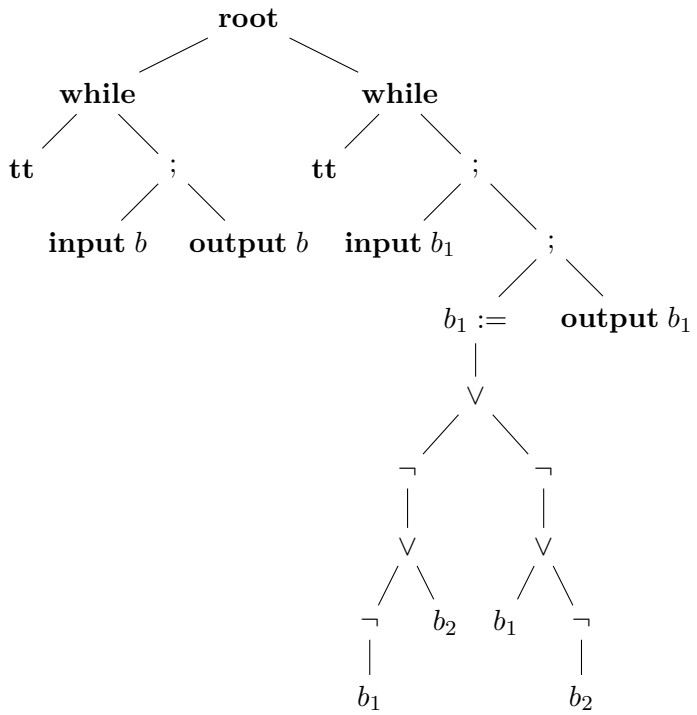

Figure 3.4: Tree representation of the distributed program in Figure 3.3

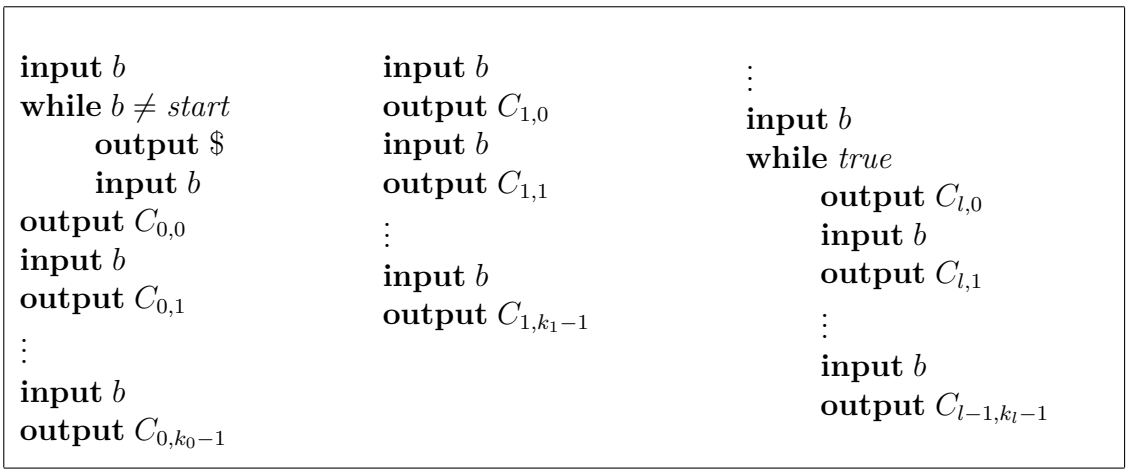

Figure 3.5: Program simulating a Turing machine. To be read column wise

3.2 Undecidability Results

3.2.1 Undecidability of Distributed Synthesis of Reactive Programs

In this section we proof that the synthesis of distributed reactive programs is undecidable. The proof is an adaption of the undecidability result in [6], which we provide in Section 2.3.1. While there the goal was to synthesize a transition system, here we synthesize programs. Our programs, however, are restricted in the memory they are allowed to use. We show that this does not affect the proof.

Lemma 3.1. *Assume we are given a halting Turing machine M. Then we can create a program with a constant amount of memory that can start at any time to output the computation of M on the empty input tape. Once the complete computation has been produced the program outputs the terminating configuration.*

Proof. We assume that the programming language can output constants not stored in variables. The programming language we use in our synthesis approach does not support this, however it is easy to add this feature by increasing the number of output labels. In Figure 3.5 we give a program outputting the computation of M on the empty input. In the first column in the loop it first reads inputs until it receives the *start*-signal, which indicates it should start to output the starting configuration C_0 of M. To output the first configuration C_0 it needs k_0 steps. In the second column, the program outputs the second configuration C_1 in k_1 steps. After producing l different configurations (for some l depending on M), it will output the terminating configuration C_l (which contains the accepting state $\perp$) of M, and keep doing so. We can construct such a program for any terminating Turing machine. Thus Pnueli's and Rosner's proof for transition systems also holds for memory restricted programs.

□

We use this result to prove the undecidability of the distributed synthesis of reactive programs.

Theorem 3.1. *The distributed synthesis of reactive programs is undecidable.*

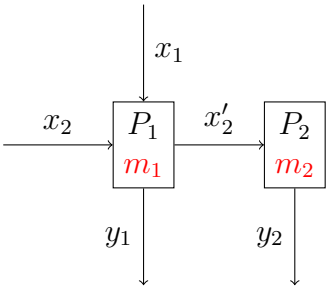

Figure 3.6: Adaption of A_0 into a hierarchical architecture

Proof. Given a Turing machine M and the architecture $\mathcal{A}_0$ we show that the specification $\psi_0 \wedge \psi_1 \wedge \psi_2$ is realizable by programs p_1, p_2 iff M halts. For transition systems, the only way to realize $\psi_0 \wedge \psi_1$ on $\mathcal{A}_0$ is to output the computation of M on the empty input. As reactive programs are not more expressive than transition systems p_1, p_2 must output the computation of M on the empty input to realize $\psi_0 \wedge \psi_1$. In Lemma 3.1 we have shown that reactive programs can output the computation of a halting Turing machine. Thus, if M halts $\psi_0 \wedge \psi_1 \wedge \psi_2$ is realizable. If M does not halt $\psi_0 \wedge \psi_1 \wedge \psi_2$ is not realizable, because p_1, p_2 will never output the accepting state $\perp$ once they start to output the computation of M. □

3.2.2 Undecidability of Distributed Program Synthesis over Hierarchical Architectures

We conjecture the distributed program synthesis over hierarchical architectures to be undecidable and in the following argue why. This undecidability is surprising, because it is decidable for transition systems, as Pnueli and Rosner showed in [13]. We are given a Turing machine M for which we want to synthesize programs iff M halts. Consider the hierarchical architecture in Figure 3.6 and note that in Lemma 3.1 we have shown that only a constant amount of memory is needed to let a program simulate a halting Turing machine. The y_i-signals output configurations of M and the x_i-signals indicate when a program should start to output configurations. Note that only the first $start_i$-signal is considered. The specification defines that

- P_1 should start to output configurations when $start_1$ is received,
- the x_2' output of P_1 has the same value as the x_2 input to P_1,
- $l = m_2 = 2^{m_1} + 1$ and
- P_2 starts to output configurations in $l + 1$ steps iff $S_2 \wedge \underbrace{\bigcirc\bigcirc \ldots \bigcirc}_{l} S_2$ is true.

The idea is that the encoding of when P_2 should start to output a configuration can not be decoded by P_1 with an amount of m_1 memory. However it can be decoded by P_2 with a memory of size m_2. To decode the $start_2$-signal P_2 only has to remember the last l inputs. If the input from l steps ago, and the input read in the previous step are both true it should start to output configurations in the next step.

The memory of P_1 is too small to store the last l inputs in its memory. However, it can read an input into memory and then do an if-then-else branching over the input read.

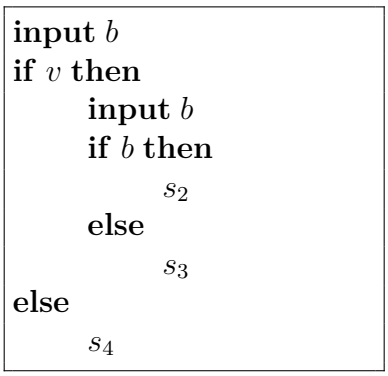

Figure 3.7: Abuse of the program counter as memory

Within the if-statement it can overwrite the memory without removing the knowledge of the last input. In Figure 3.7 we show an example, where within the outer if-statement the program can overwrite the variable b without forgetting the value it read in the first input statement. We say that it stores the input in the program counter. This argumentation stems from the fact that the program memory is restricted, but the program size is not restricted.

We think, even when P_1 abuses the program counter as memory, P_1 still can not decode the start signal. Assume the program remembers something in the program counter as seen in Figure 3.7, and then remembers again something in the program counter as in the inner if-statement. The program then has to destroy the information about the inner if-statement, before it can remove the information about the outer if-statement (in a sense the memory of the program counter works as a stack, with the exception that all information of the program counter is accessible at any time). However, we can not keep adding information to the program counter forever, as the program has to be finite. Thus P_1 can not decode the $start_2$-signal and P_1 and P_2 do not have any information about the what the other process is doing and we can force them to simulate M if the Turing machine halts.

3.3 Restricted Distributed Reactive Programs

As the decidability of distributed program synthesis is unclear (see Section 3.2.2) we decided to restrict the expressiveness of programs by setting a bound U_O^j on the number of output statements a program can have. We extend our definition of architectures with the bounds and get $\mathcal{A} = (P, P_0, E, O, \{B_1, \ldots, B_n\}, \{U_O^1, \ldots, U_O^n\})$. For restricted programs the terms program p_j can use are

$$\begin{aligned}\Sigma_j =& \{;, \mathbf{if}, \mathbf{then}, \mathbf{while}, \neg, \vee, \mathbf{tt}, \mathbf{ff}\} \cup B_j \cup \{:=_b \mid b \in B_j\} \\ & \cup \{\mathbf{input}\ \vec{b} \mid \vec{b} \in B_j^{N_I^j}\} \cup \{\mathbf{output}_r\ \vec{b} \mid \vec{b} \in B_j^{N_O^j}, 0 \leq r < U_O^j\},\end{aligned}$$

where r is considered as an ID of an output statement. The alphabet of the syntax tree which contains all programs is $\Sigma = \{\mathbf{root}\} \cup \bigcup_{j \in \{1,\ldots,n\}} \Sigma_j$. The EBNF for restricted programs becomes complicated, as we have to ensure that all programs have at most U_O^j output statements. Instead of an EBNF we define a tree automaton A_{pgm} which accepts restricted programs with correct syntax. First we define an automaton A_{pgm}^2 which accepts

trees (the trees may not represent programs with correct syntax) where the j-th subtree of root has at most U_O^j output statements. Let A^3_{pgm} be A^1_{pgm} run over restricted programs and let it ignore the output-IDs. Then we define $A_{pgm} = A^2_{pgm} \cap A^3_{pgm}$, which ensures that restricted distributed programs have correct syntax.

To keep the state space of A^2_{pgm} small we demand w.l.o.g. that in all programs respectively

- all output-IDs of output statements are different and
- the leftmost output statement has ID 0, second leftmost has ID 1 etc.

We could define A^2_{pgm} as a one-way nondeterministic automaton (with any acceptance condition), which has as state space the set of all ranges over the different U_O^j, which is in $\mathcal{O}((U_O^1)^2 + \cdots + (U_O^n)^2)$. At every node the automaton would nondeterministically separate the ranges of allowed IDs into two, where the left branch always gets the lower range. At the leafs the automaton would validate its guess. However we thought it more convenient to define A^2_{pgm} as a two-way deterministic automaton (with any acceptance condition), with statespace linear in the different U_O^j.

We describe A^2_{pgm} in more detail. A^2_{pgm} is a two-way deterministic automaton of any acceptance condition. It

- goes into all subtrees of root and there
- iterates through the leafs from left to right.
- A branch accepts when root is seen and rejects when a violation of the order or uniqueness or range of output-IDs is encountered.

Formally, $A^2_{pgm} = (\Upsilon, \Sigma_j, Q, \{q_{init}\}, \delta_{root}, \delta_{LR}, \delta_L, \delta_\emptyset, \emptyset)$ with $\Upsilon = \{L, R\} \cup \{1, \ldots, n\}$, $Q = \bigcup_{j\in\{1,\ldots,n\}}(\{j\} \times \{0, \ldots, U_O^j - 1\})$, $q_{init} = \{(1, 0)\}$ and δ we explain in several steps. The state space stores the subtree it is in and the next output-ID it allows to see. First the automaton goes into all subtrees of root and if it comes up from some branch to the root the automaton explored all leafs and thus accepts.

$$\delta_{root}(q_{init}, \mathbf{root}, U) = ((q_{init}, 1), \ldots, (q_{init}, n))$$
$$\delta_{root}((j, r), \mathbf{root}, j) = \mathit{true}$$

Then the automaton goes through all leafs from left to right.

$$\begin{aligned}\delta_{LR}((q, \sigma, U) &= (q, L)\\ \delta_{LR}(q, \sigma, L) &= (q, R)\\ \delta_{LR}(q, \sigma, R) &= (q, U)\\ \delta_L(q, \sigma, U) &= (q, L)\\ \delta_L(q, \sigma, L) &= (q, U)\end{aligned}$$

At a leaf, if the label is not an output statement the automaton goes up to go to the next leaf.

$$\delta_\emptyset(q, \sigma, U) = (q, U) \text{ if } \forall r, \vec{b}.\sigma \neq \mathbf{output}_r\ \vec{b}$$

If the label is an output statement the automaton ensures that the ID is the one it allowed, increases the ID it allows by 1 and continues. If the ID is not allowed it rejects the tree.

$$\delta_\emptyset((j, r), \mathbf{output}_{r'}\ \vec{b}, U) =$$
$$\begin{array}{ll}((j, r + 1), U) & \text{if } r = r' \text{ and } r < U_O^j\\ \mathit{false} & \text{else}\end{array}$$

and all other transitions map to *false*.

3.4 Synthesis of Restricted Distributed Reactive Programs

For the synthesis we define two two-way nondeterministic Büchi tree automata A_{sim} and $A_{non-reac}$ which together accept all distributed programs we are not interested in, where A_{sim} accepts distributed programs violating φ and $A_{non-reac}$ accepts non-reactive programs. We then complement the union of their language, and restrict the resulting language to programs, as the complement also contains trees which do not represent programs. Formally, the programs that realize the property are

$$A = \overline{A_{sim} \cup A_{non-reac}} \cap A_{pgm}.$$

We complement our two-way nondeterministic Büchi tree automata by dualizing the transition function and changing the acceptance condition from Büchi to co-Büchi. To extract a distributed program which realizes the specification we use Lemma 2.4 to create from A a one-way automaton A' that also accepts distributed program trees realizing φ. We check A' for emptiness and get a realizing distributed program, if φ is realizable. Note that A' is exponentially larger than A, because Vardi's construction uses determinization.

3.4.1 Simulator

In this section we define the simulator that checks if a distributed program violates the specification φ. This is equivalent to checking if for some input sequence from the environment the negated specification $\neg\varphi$ is satisfied. The simulator does this by interpreting the distributed program in a sequential fashion and simulating $A_{\neg\varphi}$ at the same time. Informally the simulator:

1. First interprets every program, one after another, from the beginning until an output statement. At an input statement the program reads some default values (because it is the first cycle).
2. It advances $A_{\neg\varphi}$ one step with the outputs computed (this finishes the first cycle).
3. It interprets every program from the output statement where the program was interrupted in the previous cycle until the next output statement. At an input statement the outputs from the previous cycle are read.
4. It advances $A_{\neg\varphi}$ one step with the outputs computed (this finishes the second cycle) and so on.

The simulator is defined as $A_{sim} = (\Upsilon, \Sigma, P, \{p_{init}\}, \delta_{root}, \delta_{LR}, \delta_L, \delta_\emptyset, F)$, where $\Upsilon = \{L, R\} \cup \{1, \ldots, n\}$ and $\Sigma = \{\mathbf{root}\} \cup \bigcup_{j \in \{1,\ldots,n\}} \Sigma_j$ for the program alphabets as defined in Section 3.3. The missing definitions for $P, p_{init}, \delta_{root}, \delta_{LR}, \delta_L, \delta_\emptyset, F$ are given in the following paragraphs.

State space The state space P of the simulator is

$$P : \{1, \ldots, n\} \times S \times \{0, 1\} \times Q_{\neg\varphi} \times G \times G \times \{\uparrow, \downarrow, \circlearrowleft\} \times U \times \{\mathit{fin}, \overline{\mathit{fin}}\},$$

where

- $\{1, \ldots, n\}$ defines which program we currently execute (as variables we use j),
- $S : B \to \{0, 1\}$ is the set of all variable valuations of the different programs (as variables we use s).
- $\{0, 1\}$ is used to evaluate expressions (as variables we use v),
- $Q_{\neg\varphi}$ is the set of states from the specification automaton (as variables we use q),
- G is the set of all signal valuations g with $g : O \to \{0, 1\}$. A signal valuation assigns all signals in O a value and is given to the specification automaton to advance it one step. We store two signal valuations to differentiate between an old valuation (from which input values are read) and a new valuation (to which output values are written).
- $U = \{-, 0, \ldots, U_O^1 - 1\} \times \cdots \times \{-, 0, \ldots, U_O^n - 1\}$ is the set of output IDs where programs are interrupted. The $-$ means that the program has not been interpreted yet (as variables we use pc)
- $\{\uparrow, \downarrow, \circlearrowleft\}$ defines if the automaton wants to *leave* ($\uparrow$) the current program, *enter* ($\downarrow$) the next program, or if the automaton want to *stay* ($\circlearrowleft$) and execute in the current program. As variables we use a.
- $\{\mathit{fin}, \overline{\mathit{fin}}\}$ defines the set of final states, i.e. all states, where this value is *fin* and the current state s of $A_{\neg\varphi}$ is $s \in F_{A_{\neg\varphi}}$, are final. As variables we use f.

The initial state is

$$p_{\mathit{init}} : (1, s_{\mathit{init}}, 0, q_{\mathit{init}}, g_{\mathit{init}}, g_{\mathit{init}}, \circlearrowleft, pc_{\mathit{init}}, \overline{\mathit{fin}}),$$

where

- The first 1 means that the first program to execute is p_1,
- s_{init} is the valuation where all variables have the value 0,
- the 0 will not be used and will be overwritten at the first evaluation of an expression,
- q_{init} is the initial state of $A_{\neg\varphi}$,
- g_{init} is the initial valuation for the first output produced, which will be the first input programs read. Here we assume the initial signal value to be 0,
- $\circlearrowleft$ means that we want to execute a program,
- pc_{init} is the value of the various program counters. Initially all program counters are set to -1 which means that no program has been executed yet,
- $\overline{\mathit{fin}}$ means that the state is not accepting.

The set of final states is

$$F : \{1, \ldots, n\} \times S \times \{0, 1\} \times F_{\neg\varphi} \times G \times G \times \{\uparrow, \downarrow, \circlearrowleft\} \times U \times \{\mathit{fin}\},$$

where $F_{\neg\varphi}$ is the set of final states of the specification automaton $A_{\neg\varphi}$.

Transition Function

As the transition functions are very big we consider them for different inputs separately. All transitions not explicitly defined map to *false*. The first transition the automaton takes is

$$\delta_{root}(p_{init}, \mathbf{root}, U) = (p_{init}, 1).$$

If a program reaches the root during execution this means it terminated. In this case the automaton takes the transition

$$\delta_{root}((j, s, v, q, g, g', \circlearrowleft, pc, f), \mathbf{root}, j) = true$$

and accepts.

Interprogram movement These transitions are concerned with the movement between programs. Note that we do not define these transitions for nodes with only one child, as those nodes are not relevant for interprogram movement. The movement is separated into the leaving of the current program ($\uparrow$), and the entering of the next program ($\downarrow$). The first transition handles the leaving of the current program. If the automaton is somewhere in a program and has to go to the next program it goes up.

$$\begin{aligned}
&\delta_{LR}((j, s, v, q, g, g', \uparrow, pc, f), \sigma, L) = \\
&\delta_{LR}((j, s, v, q, g, g', \uparrow, pc, f), \sigma, R) = \\
&\quad ((j, s, v, q, g, g', \uparrow, pc, \overline{fin}), U) && \text{if } \sigma \neq \mathbf{root}
\end{aligned}$$

If the automaton has to go to the next program and it is at the root. Then let $j' = (j + 1 \mod n) + 1$ be the next program to be executed. The automaton either starts executing program j' from the start, or it searches for the statement stored in $pc_{j'}$ from where the execution should continue.

$$\begin{aligned}
&\delta_{root}((j, s, v, q, g, g', \uparrow, pc, f), \mathbf{root}, j) = \\
&\quad ((j', s, v, q, g, g', \circlearrowleft, pc, \overline{fin}), j') && \text{if } pc_{j'} = - \\
&\quad ((j', s, v, q, g, g', \downarrow, pc, \overline{fin}), j') && \text{else}
\end{aligned}$$

When the automaton searches for an output statement in the program the automaton knows it only has to check the leafs. So it goes into all leafs of the tree.

$$\begin{aligned}
&\delta_{LR}((j, s, v, q, g, g', \downarrow, pc, f), \sigma, U) = \\
&\quad ((j, s, v, q, g, g', \downarrow, pc, f)), L) \vee ((j, s, v, q, g, g', \downarrow, pc, f), R)
\end{aligned}$$

At a leaf the automaton checks if the leaf is labeled by the output statement we stored. If it is, the automaton continues the execution of the program, else we evaluate to *false*.

$$\begin{aligned}
&\delta_{\emptyset}((j, s, v, q, g, g', \downarrow, pc, f), \sigma, U) = \\
&\quad ((j, s, v, q, g, g', \circlearrowleft, pc, f), U) && \text{if } \sigma = output_{pc_j} \vec{b} \text{ for some } \vec{b} \\
&\quad false && \text{else}
\end{aligned}$$

Boolean Expressions These transitions evaluate boolean expressions and are used e.g. to decide whether a while-loop should be executed. The automaton evaluates bottom up, so in the following transitions the automaton first goes down to the leftmost leaf of the expression.

$$\begin{aligned}&\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\vee,U)\\&=\delta_{L}((j,s,v,q,g,g',\circlearrowleft,pc,f),\neg,U)=((j,s,v,q,g,g',\circlearrowleft,pc,f),L)\end{aligned}$$

At a leaf the automaton remembers the value.

$$\begin{aligned}\delta_{\emptyset}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{tt},U)&=((j,s,1,q,g,g',\circlearrowleft,pc,f),U)\\\delta_{\emptyset}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{ff},U)&=((j,s,0,q,g,g',\circlearrowleft,pc,f),U)\\\delta_{\emptyset}((j,s,v,q,g,g',\circlearrowleft,pc,f),b,U)&=((j,s,s(b),q,g,g',\circlearrowleft,pc,f),U)\end{aligned}$$

When the automaton comes up to a $\neg$ it negates the result it calculated so far. When it comes up to a $\vee$, if the left child evaluated to 1 it does not evaluate the right child, else it evaluates the right child.

$$\begin{aligned}\delta_{L}((j,s,v,q,g,g',\circlearrowleft,pc,f),\neg,L)&=((j,s,1-v,q,g,g',\circlearrowleft,pc,f),U)\\\delta_{LR}((j,s,1,q,g,g',\circlearrowleft,pc,f),\vee,L)&=((j,s,1,q,g,g',\circlearrowleft,pc,f),U)\\\delta_{LR}((j,s,0,q,g,g',\circlearrowleft,pc,f),\vee,L)&=((j,s,0,q,g,g',\circlearrowleft,pc,f),R)\\\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\vee,R)&=((j,s,v,q,g,g',\circlearrowleft,pc,f),U)\end{aligned}$$

Control Flow Statements These transitions handle the control flow of the program. For the sequential operator ; we define the order of execution:

$$\begin{aligned}\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),;,U)&=((j,s,v,q,g,g',\circlearrowleft,pc,f),L)\\\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),;,L)&=((j,s,v,q,g,g',\circlearrowleft,pc,f),R)\\\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),;,R)&=((j,s,v,q,g,g',\circlearrowleft,pc,f),U)\end{aligned}$$

At while-statements, if the automaton comes down from another statement, or when it executed the loop once and comes up from the right child the automaton goes into the left child to evaluate whether the loop-condition holds.

$$\begin{aligned}&\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{while},U)\\&=\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{while},R)=((j,s,v,q,g,g',\circlearrowleft,pc,f),L)\end{aligned}$$

When the automaton sees a while-statement and it came up from the left child, it evaluated the loop-guard. Depending on the evaluation of the loop-guard it goes right into the loop body, or it goes up.

$$\begin{aligned}\delta_{LR}((j,s,1,q,g,g',\circlearrowleft,pc,f),\mathbf{while},L)&=((j,s,1,q,g,g',\circlearrowleft,pc,f),R)\\\delta_{LR}((j,s,0,q,g,g',\circlearrowleft,pc,f),\mathbf{while},L)&=((j,s,0,q,g,g',\circlearrowleft,pc,f),U)\end{aligned}$$

At if-statements, similar to while-statements, the automaton first evaluates the value of the condition in the left child. Depending on the evaluation of the expression the

automaton executes the then branch (left child of then-statement), or the else branch (right child of then-statement).

$$\begin{aligned}
\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{if},U) &= ((j,s,v,q,g,g',\circlearrowleft,pc,f),L)\\
\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{if},L) &= ((j,s,v,q,g,g',\circlearrowleft,pc,f),R)\\
\delta_{LR}((j,s,1,q,g,g',\circlearrowleft,pc,f),\mathbf{then},U) &= ((j,s,1,q,g,g',\circlearrowleft,pc,f),L)\\
\delta_{LR}((j,s,0,q,g,g',\circlearrowleft,pc,f),\mathbf{then},U) &= ((j,s,0,q,g,g',\circlearrowleft,pc,f),R)
\end{aligned}$$

When the automaton executed a branch of the then-statement, it goes up to search for the next statement. Similarly, when the automaton executed an if-statement the automaton goes up.

$$\begin{aligned}
&\delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{then},L)\\
&= \delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{then},R)\\
&= \delta_{LR}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{if},R) = ((j,s,v,q,g,g',\circlearrowleft,pc,f),U)
\end{aligned}$$

Assignment statements These transitions are concerned with the assignment of values to variables. At skip-statements the automaton only goes up.

$$\delta_{\emptyset}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{skip},U) = ((j,s,v,q,g,g'\circlearrowleft,pc,f),U)$$

At $:=_b$-statements, when coming from above, the automaton first goes into the expression and evaluates it. When the automaton comes from the left child, the automaton substitutes the value of the variable in the label with the evaluation of the expression.

$$\begin{aligned}
\delta_{L}((j,s,v,q,g,g',\circlearrowleft,pc,f),:=_b,U) &= ((j,s,v,q,g,g',\circlearrowleft,pc,f),L)\\
\delta_{L}((j,s,v,q,g,g',\circlearrowleft,pc,f),:=_b,L) &= ((j,s[b/v],v,q,g,g',\circlearrowleft,pc,f),U)
\end{aligned}$$

I/O statements These transitions are the most complex ones and handle the behavior of the I/O statements. For signals o, o', a variable b, a signal valuation g and a memory valuation s we define a function $write_{val}$ as

$$\begin{aligned}
&write_{val} : \{0,1\} \times O \times G \to G\\
&write_{val}(v,o,g)(o') = \begin{cases} v & \text{if } o' = o \\ g(o') & \text{else.} \end{cases}
\end{aligned}$$

The function updates the value of the signal o in the valuation g to the value v. Further we define a function *write* which updates the value of o under the valuation g with the value of the variable b under the valuation s. The function is defined as

$$\begin{aligned}
&write : B \times S \times O \times G \to G\\
&write(b,s,o,g)(o') = write_{val}(s(b),o,g).
\end{aligned}$$

Both functions can easily be lifted to vectors of values and vectors of variables and signals. At a statement $output_r\ \vec{b}$ the automaton updates g' with the values of the variables in $\vec{b}$ which then can be read by other programs in the next cycle. Also the automaton updates the *pc*, and initiates the switch to the next program

$$\begin{aligned}
&\delta_{\emptyset}((j,s,v,q,g,g',\circlearrowleft,pc,f),\mathbf{output}_r\ \vec{b},U) =\\
&\quad((j+1,s,v,q,g,write(\vec{b},s,signals_{out}(j),g'),\uparrow,pc[pc_j/r],f),U) \qquad \text{if } j < n,
\end{aligned}$$

where $signals_{out}(j)$ are the signals that leave program j (see Section 3.1 on page 22). If $j = n$ holds the automaton additionally to the other output transitions

- updates g' with values from the environment,
- advances the specification automaton $A_{\neg\varphi}$,
- sets the variable g to the new value of the g' variable to allow programs to read the outputs produced and
- sets the *fin*-flag, which is set to $\overline{fin}$ in the next step.

We have

$$\delta_\emptyset((j, s, v, q, g, g', \circlearrowleft, pc, f), \mathbf{output}_r\, \vec{b}, U) =$$
$$\bigvee_{val \in \{0,1\}^{N_O^0}} \bigvee_{q' \in \delta_{A_{\neg\varphi}}(q,g'')} (1, s, v, q', g''', g''', \uparrow, pc[pc_j/r], fin), U) \qquad \text{if } j = n,$$

where $g'' = write(\vec{b}, s, signals_{out}(n), g')$
and $g''' = write_{val}(val, signals_{out}(0), g'')$

At an input statement **input** $\vec{b}$ a program reads outputs produced by other programs and by the environment in the previous cycle and updates the variables in $\vec{b}$ with those outputs. For this we define a function *read* as

$$read : O \times G \times B \times S \to S$$
$$read(b, s, o, g)(b') = \begin{cases} g(o) & \text{if } b' = b \\ s(b') & \text{else.} \end{cases}$$

The function takes a signal, a variable and valuations for both and updates the variable valuation with the value of the signal. Now we can define the transitions for input statements as

$$\delta_\emptyset((j, s, v, q, g, g', \circlearrowleft, pc, f), \mathbf{input}\, \vec{b}, U) = ((j, read(\vec{b}, s, signals_{in}(j), g), v, q, g, g', \circlearrowleft, pc, f), U),$$

where $signals_{in}(j)$ is a vector of signals program j reads (see Section 3.1 on page 22).

3.4.2 The Non-Reactive Checker

The distributed non-reactive checker $A_{non-reac}$ interprets a distributed program, like A_{sim} does, but $A_{non-reac}$ does not simulate the specification automaton $A_{\neg\varphi}$. It only interprets the distributed program to determine the reachable statements. If, for some input sequence the distributed program shows non-reactive behavior the distributed program is accepted. As non-reactive behavior we define

- if the first I/O statement encountered during interpretation is not an input,
- if the I/O statements do not always alternate,
- if the program terminates or
- if eventually no I/O statement is encountered anymore.

To check the first two properties the automaton has a variable that determines the I/O statement a reactive program expects. If, during interpretation, the expectation is violated the automaton accepts. To check termination, the automaton accepts if during execution the root of the distributed program is encountered. For the last reactive property, at every I/O statement encountered (and at the start) the automaton starts a disjunctive copy that interprets the program, accepts if it never encounters an I/O statement and rejects if it does. Formally, $A_{non-reac} = (\Upsilon, \Sigma, P, \{p_{init}\}, \delta_{root}, \delta_{LR}, \delta_L, \delta_\emptyset, F)$, where $\Upsilon = \{L, R\} \cup \{1, \ldots, n\}$ and $\Sigma = \{\mathbf{root}\} \cup \bigcup_{j \in \{1,\ldots,n\}} \Sigma_j$ for the syntax tree alphabets as defined in Section 3.3. The missing definitions for $P, p_{init}, \delta_{root}, \delta_{LR}, \delta_L, \delta_\emptyset, F$ are given in the following paragraphs.

Statespace The statespace is

$$P : \{1, \ldots, n\} \times S \times \{0, 1\} \times G \times G \times \{\uparrow, \downarrow, \circlearrowleft\} \times U \times \{inp, out, \overline{io}\},$$

where the idea of *inp*, *out* is to determine which I/O statement a reactive program expects and $\overline{io}$ is a state where the automaton accepts if it never sees an I/O statement. The rest of the statespace has the same intention as in the simulator. The set of initial states is

$$p_{init} : (1, s_{init}, 0, g_{init}, g_{init}, \circlearrowleft, pc_{init}, inp)$$

which means that the expected I/O statement is an input. However, if the automaton never sees an I/O statement, then the program is not reactive and is accepted by the state with $\overline{io}$. The set of final states is

$$F : \{1, \ldots, n\} \times S \times \{0, 1\} \times G \times G \times \{\uparrow, \downarrow, \circlearrowleft\} \times U \times \{\overline{io}\}.$$

Transition Function We only define transitions for I/O statements and the first transition the automaton takes. For all other labels the transitions from the simulator are taken, and the additional values in the state space do not affect the transitions. With the first transition the automaton starts interpretation of the programs and guesses that it will never encounter an I/O statement

$$\delta_{root}(p_{init}, \mathbf{root}, U) = (1, s_{init}, 0, g_{init}, g_{init}, \circlearrowleft, pc_{init}, inp) \vee (1, s_{init}, 0, g_{init}, g_{init}, \circlearrowleft, pc_{init}, \overline{io}).$$

At an input statement program p_j reads the previous outputs form the other programs and continues interpretation and additionally, disjunctively attempts to confirm that it will never see another I/O statement.

$$\begin{aligned}
&\delta_\emptyset(j, s, v, g, g', \circlearrowleft, pc, inp), \mathbf{input}\, \vec{b}, U) \\
&\qquad = ((j, s', v, g, g' \circlearrowleft, pc, out), U) \vee ((j, s', v, g, g' \circlearrowleft, pc, \overline{io}), U), \\
&\text{where } s' = read(\vec{b}, s, signals_{in}(j), g)
\end{aligned}$$

At an output statement the automaton stores the output computed, remembers the ID of the output statement and starts to switch to the next program, as the simulator does. Additionally, the automaton guesses that it will not encounter an I/O statement during interpretation of the next program.

$$\begin{aligned}
&\delta_\emptyset((j, s, v, g, g', \circlearrowleft, pc, out), \mathbf{output}_r\, \vec{b}, U) = \\
&\qquad ((j+1, s, v, g, g'', \uparrow, pc', inp), U) \vee ((j+1, s, v, g, g'', \uparrow, pc', \overline{io}), U) \qquad \text{if } j < n \\
&\text{where } g'' = write(\vec{b}, s, signals_{out}(j), g') \\
&\text{and } pc' = pc[pc_j/r]
\end{aligned}$$

If $j = n$ holds the automaton additionally to the other output-transitions

- writes outputs from the environment to the o' variable,
- sets the valuation g to the new valuation g' to allow programs to read the outputs produced.

The simulator additionally advances $A_{\neg\varphi}$ and sets the *fin*-flag.

$$\delta_\emptyset((j, s, v, g, g', \circlearrowleft, pc, out), \mathbf{output}_r\, \vec{b}, U) = \bigvee_{val \in \{0,1\}^{N_O^0}} \Big((1, s, v, g''', g''', \uparrow, pc', inp), U) \vee (1, s, v, g''', g''', \uparrow, pc', \overline{io}), U) \Big) \quad \text{if } j = n$$

where $g'' = write(\vec{b}, s, signals_{out}(n), g')$
and $g''' = write_{val}(val, signals_{out}(0), g'')$
and $pc' = pc[pc_j/r]$

If the automaton encounters an input when a reactive program expects an output the automaton accepts, and vice versa.

$$\begin{aligned} &\delta_\emptyset(j, s, v, g, g', \circlearrowleft, pc, out), \mathbf{input}\, \vec{b}, U) \\ &= \delta_\emptyset(j, s, v, g, g', \circlearrowleft, pc, inp), \mathbf{output}_r\, \vec{b}, U) = true \end{aligned}$$

If the automaton guessed it would never encounter another I/O statement, but then reaches an I/O statement it rejects.

$$\begin{aligned} &\delta_\emptyset(j, s, v, g, g', \circlearrowleft, pc, \overline{io}), \mathbf{input}\, \vec{b}, U) \\ &= \delta_\emptyset(j, s, v, g, g', \circlearrowleft, pc, \overline{io}), \mathbf{output}_r\, \vec{b}, U) = false \end{aligned}$$

3.4.3 Further Remarks

Optimizations We can make the statespace of our automata much smaller by replacing one of the communication storages with the set $\{wrt, \overline{wrt}\}$, where a variable name for the set is w. We separate a cycle of the distributed program into an interpretation phase where $w = \overline{wrt}$, and a writing phase with $w = wrt$. During the interpretation phase the automaton only computes the output values of a program, but does not update the signal valuation g yet. This ensures that the programs use the old communicated values. After the interpretation phase the automaton starts the writing phase. In this phase the automaton searches for the output statements it remembered in the variable pc and updates the signal valuation g with the variable valuation s, which contains the outputs computed. At the output statement of the last program the automaton advances $A_{\neg\varphi}$ with the signal valuation g, and in the next cycle the automaton reads the inputs of the programs from g. For $A_{non-reac}$ we just skip the advancing of $A_{\neg\varphi}$. However, we think it is easier to understand the construction with two communication storages.

Further Work For sequential programs Madhusudan uses the power of alternation for the interpretation of programs [10]. When the automaton needs to evaluate an expression, the automaton nondeterministically guesses the value of the expression and continues interpretation with the guessed value. Conjunctively the automaton validates the guessed

value by evaluating the expression. If the guess was correct the validating branch terminates accepting. If the guess was wrong the conjunctive evaluation terminates rejecting. With this approach Madhusudan also can synthesize recursive functions, for a fixed set of function symbols and arities. When the simulator encounters a function it guesses whether the function terminates or not. If it terminates it guesses the valuation for the variables after termination and it also guesses the state of the specification automaton after it terminates. Then the simulator continues execution where it encountered the function with the guessed values and conjunctively interprets the function to validate its guess. If the function does not terminate the automaton goes to the function body and normally interprets it. It is unclear if this approach can be used in our distributed synthesis setting with restricted programs, because the state of the specification automaton depends on the behavior of all programs, which all may encounter functions.

Complexity The complexity of Madhusudan's synthesis of sequential programs is in $O(exp(|A_{\neg\varphi}| \cdot 2^{|B|}))$, where $2^{|B|}$ is the number of all possible variable valuations and $|A_{\neg\varphi}|$ is exponential in the size of φ. Let $\mathcal{A}$ be an architecture, let $2^{|\mathcal{A}|}$ be the number of all values that can be communicated on the architecture and let $|U|$ be the product of the amount of allowed output statements for the different programs, then the complexity of our construction is $O(exp(|A_{\neg\varphi}| \cdot 2^{|B|} \cdot 2^{|\mathcal{A}|} \cdot |U|))$. If B and $|U|$ are fixed then the complexity of our construction is $O(exp(|A_{\neg\varphi}| \cdot 2^{|\mathcal{A}|}))$, which is doubly exponential in the size of the specification φ and in the size of the architecture $\mathcal{A}$.

4 Implementation

We decided to implement our synthesis approach as a proof of concept. However, our implementation is not efficient enough to handle anything, but the smallest problems. In this chapter we present the main ideas of our prototype, the difficulties we encountered, our solutions for them and some small examples.

4.1 General Approach

Our approach for the implementation differs from our theoretical algorithm. In Section 3.4 we stated that we transform the two-way automaton A (which accepts the realizing programs) into a one-way nondeterministic automaton, and then test the one-way automaton for emptiness via games. The transformation algorithm suggested by Madhusudan uses Safra's determinization on an automaton with an alphabet exponential in the size of the two-way alternating automaton. We could not find a way to avoid Safra's determinization and we do not know of a tool that implements Safra's construction and can handle very large alphabets. Instead, we transform the two-way universal co-Büchi tree automaton that accepts programs realizing the specification into a SMT constraint system, similar to the approach in [5]. From a satisfying assignment to the constraint system a distributed program accepted by A can be extracted.

4.2 Constraint System

While the general bounded synthesis approach in [5] synthesizes transition systems, we synthesize trees. First we relate their approach to our setting. The states in a transition system correspond to the nodes in a tree. Further, for a state t in the transition system and an input o from the environment the successor state has to be found. To this end the authors use an uninterpreted function, for which the SMT solver has to find a satisfying assignment. In our setting the edges are labeled by directions L, R. The directions define the successor nodes, so we do not need an uninterpreted function to define which node is the successor. In our setting we only need to reason whether there exists a successor or not. We decided to use this to narrow down the search space for the SMT solver. We fix a template tree beforehand, which defines the structure of the program to be found and which may restrict the labels possible at a node. Then we let the SMT solver complete the template to a distributed program. However, this restricts the possible solutions the SMT solver can find.

Let $A = (\Upsilon, \Sigma, Q, I, \delta, F)$ be the two-way universal automaton for which we want to extract a tree from its language and let T be the template tree and let $q \in Q, v \in \Upsilon, x \in T$. Assume we have a labeling for our template tree and a run on the tree. Then, if the node

(q, υ, x) is reachable the function $\lambda^{\mathbb{B}}_{q,\upsilon} : T \to \mathbb{B}$ assigns to x the value *true*, where υ defines the direction of the previous node. Further, if $\lambda^{\#}_{q,\upsilon} : T \to \mathbb{N} \cup \{-\}$ assigns to the tree node x the value k this means on the given labeling of the template tree any run has seen at most k rejecting nodes, when it reaches the node (q, υ, x) of the run graph. Then our constraint system is

$$\forall q \in I. \lambda^{\mathbb{B}}_{q,U}(t.root),$$
$$\forall x \in T. \lambda^{\mathbb{B}}_{q,\upsilon}(x) \wedge \sigma(x) \implies \forall (q', \upsilon') \in \delta(q, \sigma, \upsilon). \lambda^{\mathbb{B}}_{q',\upsilon''}(x') \wedge \lambda^{\#}_{q',\upsilon''}(x') \rhd_q \lambda^{\#}_{q,\upsilon}(x),$$

where σ is *true* iff in node x the label σ holds, $x' = x.\upsilon'$ and $x'.\upsilon'' = x$ and $\rhd_q = >$ if $q \in F$, else $\rhd_q = \geq$. The first line defines that all initial locations are reachable. The second line defines, if the node of the run graph (q, υ, x) is reachable, and if in the node n of the template tree is labeled with σ then

- all successors (q', υ', x') (as defined by the transition function for the input q, σ, υ) are reachable and
- if q is a rejecting state, the successors have seen at least one rejecting state more. Else, they have seen at least as many as (q, υ, x).

The best way to encode the *from-direction* in the constraint system is not clear to us. We decided to have a bound for every combination of state and *from-direction*, as the *from-direction* is similar to the state of the automaton with respect to the transition function.

4.3 Concrete Implementation

In this section we describe our prototype and some of our design decisions. As SMT solver we use Z3 [11], which is a fast and modern SMT solver. One of Z3s features is a programmatic API, called Z3Py, which makes the creation of the constraints easier. Our prototype makes use of this API, however it is not difficult to instead create constraints which conform to the SMT-LIB 2 [1] standard. Our prototype takes three input parameters:

- The architecture which defines the communication, the number of variables per program and the number of output statements per program,
- the specification that should be realized and
- a template tree, which should be completed to a distributed program.

From these parameters our prototype creates a constraint system, which is then solved by the SMT solver. The SMT solver will either output *unsat*, which means that the given input tree can not be completed to a distributed program, s.t. it realizes the specification. Or it will output assignments to the labels of the nodes and a bounded annotation function, which is a certificate that the template tree with the labels is in the language of the tree automaton. This workflow is depicted in Figure 4.1.

4.3.1 Examples

As Σ_j we define all program terms that can occur in program j. As

$$\Sigma_j^{leafs} = \{B_j, \mathbf{tt}, \mathbf{ff}, \mathbf{skip}, \mathbf{output}_r\ \vec{b}, \mathbf{input}\ \vec{b}\}$$

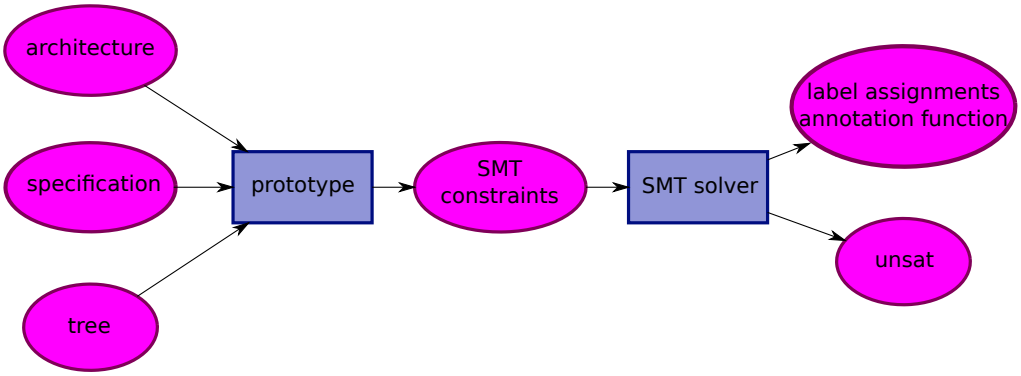

Figure 4.1: Workflow

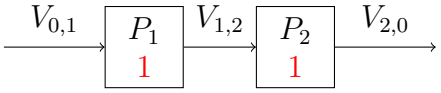

Figure 4.2: Architecture where P_1, P_2 both have one variable available

we define all terms of program j that can occur in leafs and with $-$ we label *unassigned* nodes. We visualize the template tree as a tree, where the nodes are labeled by sets. The nodes represent the fixed structure of the program and the sets of labels represent the labels the template allows. Our examples were run on an Ubuntu 12.04 machine with an Intel Quad Core with 2.6 GHz and we set the timeout to 60 minutes. We always used the optimizations mentioned in Section 4.3.2, where we chose 5 as the length of our bit-vectors. In all of our examples we use the architecture from Figure 4.2.

Example 4.1. On the specification $\square(V_{0,1} \implies \bigcirc\lozenge V_{2,0}) \wedge \square(\neg V_{0,1} \implies \bigcirc\lozenge \neg V_{2,0})$ over the architecture in Figure 4.2 and the template tree shown in Figure 4.3a our prototype needed 3:20 minutes (mm:ss) and 1.04 GB of RAM to complete the template to the distributed program seen in Figure 4.3b.

Example 4.2. The specification $\square V_{1,2}$ on our architecture and the template tree from the previous example shown in Figure 4.3a is not realizable, because the template gives just enough space to define a distributed reactive program. However, the space is not sufficient to allow any assignments within the reactive program. Our prototype needed 49 seconds and 0.49 GB to find that the specification is unrealizable on the given template.

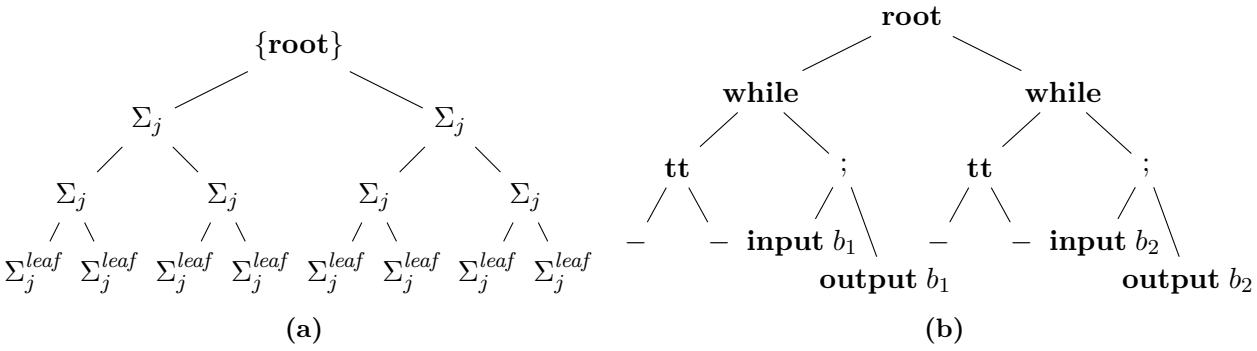

Figure 4.3: Left is a template tree where the root-label is fixed and all other nodes have arbitrary labels. Right is a distributed program created from the template tree

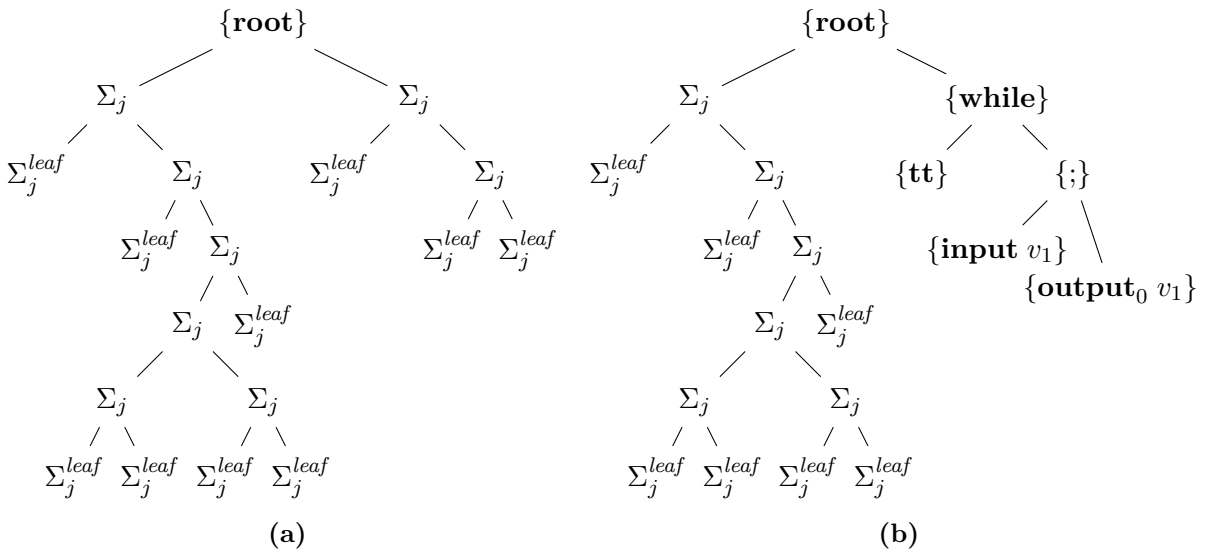

Figure 4.4: The left template tree does not restrict any labels. The right template tree fixes a program for P_2

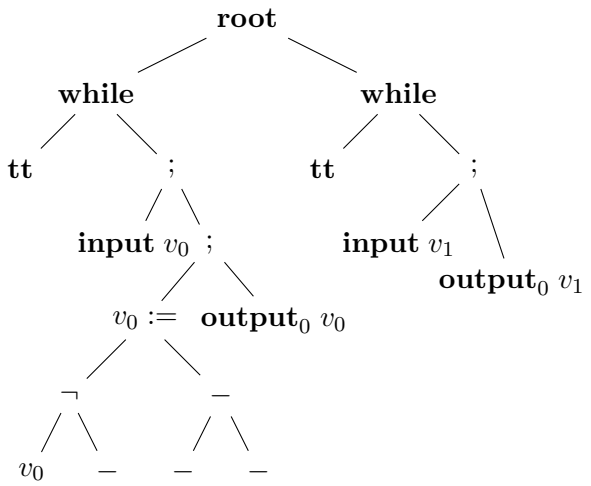

Figure 4.5: Distributed program created from the templates in Figure 4.4

Example 4.3. The specification $\square(V_{0,1}\ XOR \bigcirc\bigcirc V_{2,0})$ on our architecture with the template Figure 4.4a needed 20:04 minutes and used 2.56 GB. The synthesized programs are shown in Figure 4.5.

This example makes apparent the necessity to have some idea about how the program should look. If we choose the tree for P_1 to be a full binary tree of height 6 the synthesis takes more than 60 minutes. This is wasteful, considering we often want the first statements to be a while-true loop where all children of the node labeled with **tt** are unassigned. If we fix the children of the root-node from Figure 4.4a to be while-true loops our prototype needs 14:41 minutes to complete the program. If we take template tree Figure 4.4b, where one program is fixed already, our prototype takes 7:28 minutes to find the the distributed program in Figure 4.5. This is interesting for program completion, where we are given a partial implementation.

4.3.2 Optimizations and Heuristics

We can reduce the time the SMT solver needs for solving the constraint system by variations in the encoding or by reduction of the search space.

Bound Encoding In our constraints we define the values of the bounds as integers. However, if we restrict the maximal value of the bounds, we can use fixed-size bit-vectors to encode the values. For small sizes bit-vector arithmetic is faster than integer arithmetic. On the other hand we lose completeness. The required size depends on the number of visits to rejecting states.

Label Restrictions We disallow the following label assignments to narrow down the search space. The restrictions given do not affect completeness. Let ψ be an expression then of the equalities given we do not need any of the statements on the left side: $\neg\neg\psi \equiv \psi$, $\neg\mathbf{ff} \equiv \mathbf{tt}$, $\neg\mathbf{tt} \equiv \mathbf{ff}$, $\mathbf{tt} \vee \psi \equiv \mathbf{tt}$, $\mathbf{ff} \vee \psi \equiv \psi$, $\psi \vee \mathbf{tt} \equiv \mathbf{tt}$, $\psi \vee \mathbf{ff} \equiv \psi$. Further, let s, s_1, s_2 be program statements then the following statements can be disallowed: $\mathbf{while}\,\mathbf{ff}\{s\}$, $\mathbf{while}\,\psi\{\mathbf{skip}\}$ and these as well $\mathbf{if}\,\mathbf{tt}\,\mathbf{then}\,s_1\mathbf{else}\,s_2$ and $\mathbf{if}\,\mathbf{ff}\,\mathbf{then}\,s_1\mathbf{else}\,s_2$.

5 Conclusion

In this thesis we proved the undecidability of the synthesis of distributed reactive programs. Also, we gave an argument why we believe that distributed program synthesis over hierarchical architectures is undecidable, even though the distributed synthesis of transition systems is always decidable over hierarchical architectures. Because the decidability of distributed synthesis of reactive programs was uncertain, we parametrized the programming language with bounds on the number of output statements the different programs in the distributed system may have at most. With this restriction, the distributed synthesis problem turned out to be decidable for all architectures with a complexity similar to the case of sequential program synthesis. Further, we also implemented our approach as a proof of concept.

5.1 Further Work

Our argument for the undecidability of some architectures motivates the deeper research of the decidability of distributed synthesis of reactive programs and of the complexity of the decidable problems. Also, our solution to the distributed synthesis problem for restricted programs does not allow to synthesize programs with recursive functions. The questions remains if and how recursive functions can be supported in our approach to the synthesis of distributed reactive programs. At last, our implementation only can handle small specifications. It remains open how our synthesis procedure can be implemented more efficiently.

Bibliography

[1] C. Barrett, A. Stump, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org.

[2] B. Brütsch. "Synthesizing Structured Reactive Programs via Deterministic Tree Automata". In: *Electronic Proceedings in Theoretical Computer Science* 112 (2013), pp. 107–113.

[3] J. Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 5.7 (1960), pp. 66–92.

[4] T. Cachat. "Two-Way Tree Automata Solving Pushdown Games". In: *Automata, Logics, and Infinite Games*. Vol. 2500. 2001, pp. 303–317.

[5] B. Finkbeiner and S. Schewe. "Bounded Synthesis". In: *Automated Technology for Verification and Analysis*. 2007.

[6] B. Finkbeiner and S. Schewe. "Uniform Distributed Synthesis". In: *Symposium on Logic in Computer Science* (2005), pp. 321–330.

[7] R. Gerth et al. "Simple On-the-fly Automatic Verification of Linear Temporal Logic". In: *Protocol Specification Testing and Verification*. 1995, pp. 3–18.

[8] O. Kupferman and M. Vardi. "Synthesis with Incomplete Informatio". In: *Advances in Temporal Logic* (2000), pp. 109–127.

[9] O. Kupferman and M. Vardi. "Synthesizing Distributed Systems". In: *Symposium on Logic in Computer Science* (2001), pp. 389–398.

[10] P. Madhusudan. "Synthesizing Reactive Programs". In: *Computer Science Logic* 2011 (2011), pp. 428–442.

[11] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. 2008, pp. 337–340.

[12] A. Pnueli. "The Temporal Logic of Programs". In: *Foundations of Computer Science*. 1977, pp. 46–57.

[13] A. Pnueli and R. Rosner. "Distributed Reactive Systems are Hard to Synthesize". In: *Proceedings of the 1990 IEEE Symposium on the Foundations of Computer Science* (1990).

[14] A. Pnueli and R. Rosner. "On the Synthesis of an Asynchronous Reactive Module". In: *Automata, Languages and Programming* (1989).

[15] M. O. Rabin. "Decidability of Second-Order Theories and Automata on Infitite Trees". In: *Transactions of the American Mathematical Society* 141 (1969), pp. 1–35.

[16] S. Schewe and B. Finkbeiner. "Synthesis of Asynchronous Systems". In: *Logic-Based Program Synthesis and Transformation* 4407 (2007), pp. 127–142.

[17] W. Thomas. “Automata on Infinite Objects”. In: *Handbook of Theoretical Computer Science*. Vol. B. 1990, pp. 133–191.

[18] M. Vardi. “Reasoning about the past with two-way automata”. In: *Automata, Languages and Programming* (1998).