

Analyzing the Communication Behavior of LOOP+ ω Programs

Rheinisch–Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2

Master Thesis

Benjamin Lucien Kaminski

Reviewers:

apl. Prof. Dr. Thomas Noll

Prof. Dr. Ir. Joost-Pieter Katoen

Registration Date: 16. July 2013

Submission Date: 23. September 2013

Statement / Erklärung

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

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

Aachen, 23. September 2013

Benjamin L. Kaminski

Abstract

We introduce the model programming language $\text{LOOP}+\omega$ which represents a class of communicating programs that consist of non-terminating main loops and terminating inner loops. The inner loops of $\text{LOOP}+\omega$ programs are even required to satisfy the more strict notion of primitive recursiveness.

We study decidability questions relating to the communication behavior of $\text{LOOP}+\omega$ programs and show that virtually every interesting decision problem remains undecidable despite the restriction of certainly non-terminating main loops and primitive recursive, thus certainly terminating, inner loops. We are even able to prove that a variant of the powerful Theorem of Rice is applicable to $\text{LOOP}+\omega$ programs.

In addition we present a method by means of which it is possible to obtain a tractable ω -regular over-approximation of a $\text{LOOP}+\omega$ program's communication behavior and we show how we are able to analyze a network of communicating $\text{LOOP}+\omega$ programs with the aid of this over-approximation.

Kurzfassung

Wir präsentieren die Modellprogrammiersprache $\text{LOOP}+\omega$, die eine Klasse von kommunizierenden Programmen, die aus einer nicht-terminierenden Hauptschleife und terminierenden inneren Schleifen aufgebaut sind, darstellt. Die inneren Schleifen unterliegen sogar der noch strikteren Einschränkung, dass sie primitiv rekursiv sind.

Wir untersuchen Entscheidungsfragen, die das Kommunikationsverhalten solcher $\text{LOOP}+\omega$ -Programme betreffen und werden zeigen, dass beinahe alle interessanten Entscheidungsprobleme trotz der Einschränkung von garantiert nicht-terminierenden Hauptschleifen und primitiv rekursiven, daher garantiert terminierenden, inneren Schleifen unentscheidbar bleiben. Es ist uns sogar möglich zu beweisen, dass eine Variante des mächtigen Satzes von Rice auf $\text{LOOP}+\omega$ -Programme anwendbar ist.

Des Weiteren zeigen wir eine Methode auf, mit deren Hilfe man immerhin eine ω -reguläre, und damit handhabbare, Überapproximation des Kommunikationsverhaltens eines $\text{LOOP}+\omega$ -Programmes erhält, und zeigen auch, wie wir mit Hilfe einer solchen Überapproximation ein Netzwerk von kommunizierenden $\text{LOOP}+\omega$ -Programmen analysieren können.

Abstrakt

Prezentujeme modelový programovací jazyk $\text{LOOP}+\omega$ představující třídu programů, které se skládají z neterminující hlavní smyčky a terminujících vnitřních smyček. Tento fakt, že vnitřní smyčky jsou primitivně rekurzivní, představuje ještě přísnější omezení.

Zkoumáme otázky schopnosti rozhodování týkající se komunikačních vlastností těchto programů a demonstrujeme, že téměř všechny zajímavé rozhodovací úlohy navzdory omezením zaručeně neterminujících hlavních smyček a primitivně rekurzivních, proto zaručeně terminujících, vnitřních smyček zůstávají nerozhodnuty. Byli jsme dokonce schopni dokázat, že jedna varianta mocného Rice teorému je aplikovatelná na programy $\text{LOOP}+\omega$.

Kromě toho prezentujeme metodu, pomocí které lze obdržet w -regulární nadaproximaci komunikačního chování $\text{LOOP}+\omega$ programu a demonstrujeme také, jak můžeme pomocí této nadaproximace analyzovat síť komunikujících $\text{LOOP}+\omega$ programů.

Acknowledgments

Many people deserve to be thanked! First and foremost I would like to thank my supervisor Thomas Noll for his kind supervision of my master thesis and the uncomplicated and very pleasant collaboration.

I also want to thank my Mum and my Dad and Stefanie for their loving support not only regarding the time of writing my master thesis, and my friend Lena for proofreading a greater part of this thesis.

Last but certainly not least, I want to thank my fellow students Johannes, Russ and Philipp for their support and friendship throughout our joint years of studies.

Contents

1	Introduction	1
2	Preliminaries	3
2.1	Alphabets, Words and Languages	3
2.1.1	Regular Languages	4
2.1.1.1	Finite Automata	4
2.1.1.2	Regular Expressions	6
2.1.1.3	Closure Properties and Decision Problems	7
2.1.1.4	Lemmas for Regular Languages	8
2.1.2	Context-Free Languages	9
2.1.3	Context-Sensitive Languages	9
2.1.4	Recursively Enumerable Languages	10
2.2	ω -Words and ω -Languages	10
2.2.1	ω -Regular Languages	11
2.2.1.1	Büchi Automata	12
2.2.1.2	Characterization by Means of the ω -Power	14
2.2.1.3	Closure Properties and Decision Problems	14
2.2.2	ω -Context-Free Languages	14
2.2.3	Remarks on Context-Sensitive and Recursive ω -Languages	15
3	Syntax and Semantics of LOOP+ω Programs	17
3.1	Ordinary LOOP Programs	17
3.1.1	Syntax of LOOP Programs	17
3.1.2	Semantics of LOOP Programs	18
3.2	Message Passing	20
3.2.1	Syntax of Message Passing	20

3.2.2	The I/O–Language	21
3.2.2.1	Alphabet of the I/O–Language	21
3.2.2.2	Concatenations of Words and Configurations	22
3.2.3	Semantics of Message Passing	23
3.3	ω –Loops	25
3.3.1	Syntax of ω –Loops	25
3.3.2	Semantics of ω –Loops	25
3.4	LOOP+ ω Programs	26
3.4.1	Syntax of LOOP+ ω Programs	26
3.4.2	Semantics of LOOP+ ω Programs	27
3.4.3	I/O–Languages of LOOP+ ω Programs	28
3.5	Syntactic Sugar	30
4	Lemmas and Corollaries on Configurations and Languages	31
4.1	Endomorphic Property of the Concatenation of Words and Configurations	33
4.2	Homomorphic Property of the Application of LOOP or LOOP+ ω Semantics	33
4.3	Prefix Preservation by the Application of LOOP or LOOP+ ω Semantics	37
4.4	Miscellaneous Properties of LOOP– and LOOP+ ω –Generated Languages	37
4.5	Reactiveness of LOOP– and LOOP+ ω –Generated Languages	40
5	Decision Problems for $Loop_\omega$	43
5.1	Recursive Functions	44
5.1.1	WHILE Programs	45
5.1.2	The Halting Problem	46
5.1.3	The Kleene Normal Form Theorem	46
5.2	Expressiveness of LOOP+ ω	47
5.2.1	Simulation of WHILE–Programs in LOOP+ ω	48
5.2.2	Enumeration of all Recursively Enumerable Sets in LOOP+ ω	49
5.3	Rice’s Theorem for LOOP+ ω Programs	52
5.4	The Word Problem	57
5.5	The Emptiness Problem and the Intersection Problem	57
5.6	The Equivalence and the Inclusion Problem	60
5.7	The Regularity Problem	62
5.8	The Pulse–Distance Boundedness Problem	64
5.9	The ω –Power Problem	66

6	Over-Approximation of LOOP+ω-Generated Languages	69
6.1	ω -Regular Over-Approximation	69
6.2	Remarks on ω -Context-Sensitive Over-Approximations	78
7	Analysis of Networks of Communicating LOOP+ω Programs	79
7.1	Synchronization of Languages	79
7.1.1	Synchronization of ω -regular Languages	82
7.1.2	Synchronization of ω -Context-Free Languages	84
7.2	Interleaving Languages of Communicating Programs	85
7.3	Channel-Guard Languages	86
7.4	Fairness-Ensuring Languages	90
7.5	Application: Checking for Deadlocks	91
8	Conclusion	93
8.1	Outlook	93
8.2	Summary	94
A	Appendix — Omitted Proofs	95
A.1	Proof of Lemma L.4.2	95
A.2	Proof of Lemma L.4.5	98
A.3	Alternative Proof of Lemma L.5.6	102
	Nomenclature	107
	Bibliography	111

1

Introduction

Verification of software is becoming more and more important due to the increasing usage of complex software in systems relevant to safety and security. Increasing complexity of such software systems pushes manual debugging and testing to its limits and calls for automated application of formal methods which allow for verification of software with the reliability of a mathematical proof.

An especially difficult field of software verification is the verification of distributed systems due to the concurrent and interdependent nature of the participating processes. A particular interest lies in the communication behavior of a distributed system. In terms of their communication behavior the participating processes or rather programs can often be seen as black boxes: It is not important exactly *how* a program produces its reactions to other program's inputs but rather *what* the reaction to different inputs is. Such distributed systems can be modeled e.g. by communicating automata or message sequence charts. In this thesis, however, we choose the approach of modeling the programs in a distributed system by a tailor-made programming language called LOOP+ ω which allows for communication with other programs. Even though the model of communicating automata for instance is already Turing-complete [BZ83], we feel that a model programming language is more tangible and closer to real world examples. In the following, we want to motivate the idea behind introducing the new model language LOOP+ ω , rather than using an existing model language like WHILE or GOTO:

The KleeNet project for scalable symbolic execution of distributed systems [SLA⁺10, SDK⁺11] is an example of applying formal methods for software verification. During its development the KleeNet tool was typically applied to small implementations of protocols for communicating sensor nodes. These programs usually consist of an infinite main loop which polls for communication events or reads measurements from the node's sensors and processes them if necessary. A major difficulty of applying symbolic execution to such infinite loops is that the resulting execution tree is inevitably of infinite size and thus the analysis of the program never terminates. The symbolic execution method of KleeNet, however, does *not* take the fact into account

that the main loop is *guaranteed to not terminate*. In this master thesis we thus want to study whether we can *exploit the guaranteed non-termination of the main loop*, for achieving better analysis results for such programs. Since in this thesis we only focus on the analysis of programs with infinite main loops, we introduce the model language $\text{LOOP}+\omega$ to model exactly this class of programs.

The thesis is structured as follows: After we will have recalled some basics of formal languages and ω -languages in Chapter 2, we will introduce the novel programming language $\text{LOOP}+\omega$ in Chapter 3. $\text{LOOP}+\omega$ is supposed to model a class of programs with infinite main loops and the restriction of primitive recursive, thus terminating, inner loops. We will present the syntax of $\text{LOOP}+\omega$ programs as well as their semantics in terms of I/O-languages which represent the communication behavior of the programs. In Chapter 4 we will prove some properties for $\text{LOOP}+\omega$ programs and $\text{LOOP}+\omega$ -generated languages which are needed for our studies of the expressiveness of $\text{LOOP}+\omega$ in Chapter 5. In that chapter we will learn that, despite the restriction of certainly terminating inner loops and certainly non-terminating main loops, virtually every interesting decision problem for $\text{LOOP}+\omega$ programs remains undecidable, which gives a negative answer to the question whether we can improve on the analysis of distributed systems by restricting the participating programs to the class of $\text{LOOP}+\omega$ programs. We will even go as far as to prove that a variant of the powerful Theorem of Rice [Ric53] applies to $\text{LOOP}+\omega$ programs. In the face of undecidability issues, we will show in Chapter 6 how we can at least over-approximate a $\text{LOOP}+\omega$ program's communication behavior in a way so that the over-approximation is tractable for further analysis. In Chapter 7 we will then show how we can synchronize and interleave the over-approximations of the languages of multiple communicating $\text{LOOP}+\omega$ programs in order to analyze a whole network of communicating programs. Finally, we will give an outlook on further improvements of the presented method as well as a summary of this master thesis.

2

Preliminaries

Even though we consider the reader to be familiar with the concept of formal languages and ω -languages in general, we give a short overview of the classes of languages we will be dealing with in this thesis. In this chapter, we shall also introduce the notations we will use. We will start off by giving an overview of the basic terms in formal language theory, followed by an overview of the classes of languages in the so called Chomsky–Hierarchy. Thereafter we will give an overview on their counterparts in the domain of ω -languages.

2.1 Alphabets, Words and Languages

An **alphabet** Σ is a finite set of symbols, e.g. $\Sigma = \{a, b, c\}$. A **word** w over an alphabet Σ is a finite sequence of symbols from the alphabet Σ , e.g. $w = aabccc$. The empty sequence or **empty word** we denote by ε . The **length of a word** w is denoted by $|w|$. The **set of all words over the alphabet** Σ is denoted by Σ^* . We denote by $w[i]$ the i^{th} **symbol of the word** w (starting with the first symbol $w[1]$).

A **language** L is a (possibly infinite) set of words $L \subseteq \Sigma^*$, e.g. $L = \{\varepsilon, a, aabccc, aaabccc, abc\}$ or $L = \{b, ab, aab, aaab, aaaab, \dots\}$. The set of all languages is consequently given by $\mathcal{P}(\Sigma^*)$, where $\mathcal{P}(\cdot)$ denotes the **powerset operator**.

Two words $w, v \in \Sigma^*$ can be **concatenated** by the **\cdot -operator** to generate a new word, e.g. $a \cdot aabccc = aabccc$ or $abc \cdot \varepsilon = abc$. Analogously, languages can be concatenated, too: The concatenation $L_1 \cdot L_2$ of two languages L_1 and L_2 results in a new language containing every possible concatenation of a word in L_1 with a word in L_2 , i.e.

$$L_1 \cdot L_2 = \{w \cdot v \mid w \in L_1 \text{ and } v \in L_2\}.$$

We write $w \cdot L$ as an abbreviation for $\{w\} \cdot L$.

For any language $L \subseteq \Sigma^*$, the **Kleene–star** L^* of a language contains all finite concatenations of the language L with itself. It is hence the union over all i -fold concatenations of L with itself for all $i \in \mathbb{N}$, i.e.

$$L^* = \bigcup_{i \in \mathbb{N}} L^i$$

We write L^+ as an abbreviation for $L \cdot L^*$ and call L^+ the **positive closure of L** . Note that in particular $\varepsilon \in L^*$ for all languages L , since $\varepsilon \in \{\varepsilon\} = L^0 \subseteq L^*$.

2.1.1 Regular Languages

The first class of languages we recall is the class of *regular* languages $\mathcal{R}eg$. It is the lowest class in the Chomsky–Hierarchy. The regular languages have been well–studied for decades and there are hence various different equivalent characterizations of the regular languages, including logical definability in FO+MTC (First Order Logic with Monadic Transitive Closure), FO+MLFP (First Order Logic with Monadic Least Fixed Point) and MSO (Monadic Second Order Logic) [Pot94, Büc60]. Another very well–studied characterization of the regular languages is by means of their syntactic monoid which was studied — amongst others — by Marcel–Paul Schützenberger, who studied in particular the subclass of star–free languages [Sch65]. The two best–known characterizations of the regular languages are, however, by means of finite automata and by means of regular expressions. As we will need these two concepts in particular in this thesis, we will briefly recall them.

2.1.1.1 Finite Automata

We start with the concept of finite automata. Finite automata are theoretical devices — or machines — that accept words. The term “finite” refers to the finite amount of states the automaton can be in. In other words, the amount of memory that these devices are equipped with is finite. Consequently another term used for such devices is *finite state machine*. There are deterministic and non–deterministic versions of these machines. We first recall the definition for the deterministic ones:

Definition D.2.1 (DETERMINISTIC FINITE AUTOMATA):

A \mathfrak{A} is a quintuple $\mathfrak{A} = (Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of states,

Σ is a finite alphabet,

$\delta: Q \times \Sigma \rightarrow Q$ is a total transition function,

$q_0 \in Q$ is the initial state, and

$F \subseteq Q$ is the set of final states.

The **run ρ_w of \mathfrak{A} on a word w** is a sequence of states $\rho_w \in Q^{|w|+1}$ in which $\rho_w[1] = q_0$ and in which for all $i \in \{1, \dots, |w|\}$ it holds that

$\delta(\rho_w[i], w[i]) = \rho_w[i + 1]$. The run ρ_w of \mathfrak{A} on a word w is called an **accepting run**, if $\rho_w[|w| + 1] \in F$.

The automaton \mathfrak{A} **accepts** w , if the run of \mathfrak{A} on w is accepting, otherwise \mathfrak{A} **rejects** w . The **language $L(\mathfrak{A})$ recognized by a DFA \mathfrak{A}** is the set of all words which are accepted by \mathfrak{A} , hence $L(\mathfrak{A}) = \{w \in \Sigma^* \mid \mathfrak{A} \text{ accepts } w\}$.

We say that a language L is **DFA-recognizable**, if there exists a DFA \mathfrak{A} such that $L(\mathfrak{A}) = L$.

Note that, since δ is a *function* and furthermore δ is a *total function*, for every word w there exists exactly *one unique* run ρ_w of a DFA \mathfrak{A} on w . Hence the run ρ_w of \mathfrak{A} on w is *determined*.

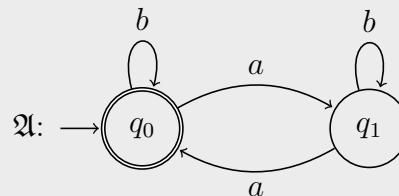
Example E.2.1 (DETERMINISTIC FINITE AUTOMATA):

The DFA $\mathfrak{A} = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_0\})$ with δ given by

δ	a	b
q_0	q_1	q_0
q_1	q_0	q_1

has two states q_0 and q_1 , accepts words over the alphabet $\{a, b\}$ as input, initially starts in state q_0 and accepts a word w if the run of \mathfrak{A} (with respect to the transition function δ) ends in the final state q_0 . \mathfrak{A} recognizes the language $L = \{w \in \{a, b\}^* \mid \text{The number of } a\text{'s in } w \text{ is even}\}$.

Alternatively to the cumbersome definition as given above, the DFA \mathfrak{A} can also be visualized and thereby be described *completely* by the following figure:



The (unique) run of \mathfrak{A} on the word $aababa$ is $q_0q_1q_0q_0q_1q_1q_0$. This run is accepting since the last state q_0 of this run is in the set of final states $\{q_0\}$. Thus \mathfrak{A} accepts $aababa$.

The (unique) run of \mathfrak{A} on the word $aabab$ is $q_0q_1q_0q_0q_1q_1$. This run is *not* accepting since the last state q_1 of this run is not in the set of final states $\{q_0\}$. Thus \mathfrak{A} rejects $aabab$.

As mentioned before, there exists the seemingly more general concept of non-deterministic finite automata (NFA). NFA allow a transition *relation* Δ instead of a transition function δ . A run of a NFA is then defined analogously to the run of a DFA but with respect to the transition relation instead of a function. Hence, there might exist *multiple* or even *no* runs of a NFA on a given word w . The acceptance condition of a NFA is consequently the following: A NFA \mathfrak{A} accepts a word w if there *exists* an accepting run of \mathfrak{A} on w .

There is also a seemingly even more general concept of NFA with ε -moves (ε -NFA). ε -NFA are NFA which additionally allow for spontaneous state transitions. An important result due to Michael O. Rabin and Dana S. Scott is, however, that the expressiveness of NFA and ε -NFA is the same as the expressiveness of DFA, i.e. all three automata accept precisely the regular languages [RS59]. So given any (ε -)NFA \mathfrak{A} one can construct an equivalent DFA \mathfrak{A}' with $L(\mathfrak{A}) = L(\mathfrak{A}')$. We will recall later that this is not the case for regular languages over infinite words.

Another notable result due to the Myhill–Nerode Theorem [Myh57, Ner58] is that for each regular language L there exists an (up to isomorphism) unique minimal DFA \mathfrak{A}_L with $L(\mathfrak{A}_L) = L$. This minimal DFA \mathfrak{A}_L of a regular language L is usually called the canonical DFA of L .

2.1.1.2 Regular Expressions

The second formalism we examine for characterizing the regular languages is the notion of regular expressions, first introduced by Stephen C. Kleene [Kle56]. The set of regular expressions is defined as follows:

Definition D.2.2 (REGULAR EXPRESSIONS):

The **set of regular expressions** over an alphabet Σ is defined inductively as the smallest set, such that

- any symbol $a \in \Sigma$ is a regular expression,
- if both r_1 and r_2 are regular expressions, then $r_1 + r_2$, $r_1 \cdot r_2$ and r_1^* are regular expressions as well.

Instead of $r_1 \cdot r_2$ we sometimes write $r_1 r_2$ for short.

The **language $L(\cdot)$ of a regular expression** is defined inductively as follows:

$$\begin{aligned} L(a) &= \{a\} \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r_1 \cdot r_2) &= L(r_1) \cdot L(r_2) \\ L(r^*) &= L(r)^* \end{aligned}$$

In his seminal paper Kleene showed that the characterization of regularity by means of finite automata agrees with the characterization of regularity by means of regular expressions [Kle56]. For that reason we say that a language L is **regular**, if a DFA, NFA or ε -NFA \mathfrak{A} with $L(\mathfrak{A}) = L$ or if a regular expression r with $L(r) = L$ exists.

Example E.2.2 (REGULAR EXPRESSIONS):

The language from Example E.2.1 is effectively given by the regular expression

$$(b + a b^* a)^*,$$

for it holds that $L = \{w \in \{a, b\}^* \mid \text{The number of } a\text{'s in } w \text{ is even}\} = L((b + a b^* a)^*)$.

The word *aababa* is in $L((b + a b^* a)^*)$, since $aababa = ab^0a \cdot b \cdot ab^1a$.

The word *aabab* on the other hand does not match the regular expression at hand, as it lacks an *a*.

2.1.1.3 Closure Properties and Decision Problems

The regular languages are a so to say a very “benign” class of languages because they satisfy many closure properties and because many important decision problems for them are decidable with reasonable time complexity.

We shall first recall the general definitions of some closure properties we are interested in: Let $\mathcal{L} \subseteq \mathcal{P}(\Sigma^*)$ be a class of languages (e.g. $\mathcal{L} = \mathcal{R}eg$) and let L_1 and L_2 be any two languages in \mathcal{L} . We say that \mathcal{L} is **closed under**

- **union**, if $L_1 \cup L_2 \in \mathcal{L}$,
- **intersection**, if $L_1 \cap L_2 \in \mathcal{L}$,
- **complementation**, if $\Sigma^* \setminus L_2 \in \mathcal{L}$,
- **concatenation**, if $L_1 \cdot L_2 \in \mathcal{L}$,
- **Kleene-closure**, if $L_1^* \in \mathcal{L}$.

Generally speaking, we say that a class of languages \mathcal{L} is closed under a k -ary operation f , if for any $L_1, \dots, L_k \in \mathcal{L}$ it holds that $f(L_1, \dots, L_k) \in \mathcal{L}$. It is quite an easy exercise to prove the following:

Remark R.2.1 (CLOSURE PROPERTIES OF $\mathcal{R}eg$ [HMU02]):

$\mathcal{R}eg$ is closed under union, intersection, complementation, concatenation and Kleene-closure.

Next we recall some basics of decision problems: We generally say that a language L is **effectively given**, if it is given by means of an accepting or generating device or formalism \mathfrak{A} which accepts or generates this language, so $L(\mathfrak{A}) = L$. A regular language for instance is, as mentioned before, effectively given by a DFA or a regular expression. Furthermore, we say that a problem P is **decidable** for a class of languages \mathcal{L} , if there exists an algorithm \mathfrak{A}_P such that for any language $L \in \mathcal{L}$ *effectively given* the question whether L satisfies the problem statement of P can be answered by \mathfrak{A}_P correctly for every input in finite time.

Now let \mathfrak{A} and \mathfrak{B} be generating or accepting devices and let $L(\mathfrak{A})$ and $L(\mathfrak{B})$ be the languages effectively given by \mathfrak{A} and \mathfrak{B} , respectively. Then a few decision problems which are of particular interest in verification of software are:

- The **Word Problem**: Is a given word $w \in L(\mathfrak{A})$?
- The **Emptiness Problem**: Does it hold that $L(\mathfrak{A}) = \emptyset$?
- The **Inclusion Problem**: Does it hold that $L(\mathfrak{A}) \subseteq L(\mathfrak{B})$?
- The **Equivalence Problem**: Does it hold that $L(\mathfrak{A}) = L(\mathfrak{B})$?

All of the above decision problems are decidable for the class of regular languages, so we have:

Remark R.2.2 (DECIDABILITY OF PROBLEMS FOR $\mathcal{R}eg$ [HMU02, MS72]):

The Word Problem, the Emptiness Problem, the Inclusion Problem and the Equivalence Problem are decidable for $\mathcal{R}eg$.

That means that if a regular language L is effectively given, say by a DFA \mathfrak{A}_L with $L(\mathfrak{A}_L) = L$, one could for instance decide whether L is finite or empty just by examining the automaton \mathfrak{A}_L .

2.1.1.4 Lemmas for Regular Languages

In the upcoming chapters we will sometimes need necessary and sufficient properties for proving whether a certain language is regular or not. A quite trivial sufficient property for a language to be regular is that the language is finite:

Lemma L.2.1 (REGULARITY OF FINITE LANGUAGES):

Every finite language is regular.

Proof: Let $L = \{w_1, \dots, w_k\} \subset \Sigma^*$ be any finite language. Then L is regular, as the regular expression $w_1 + \dots + w_k$ describes L :

$$\begin{aligned}
 & L(w_1 + \dots + w_k) \\
 = & L(w_1) \cup \dots \cup L(w_k) & \text{(D.2.2)} \\
 = & \{w_1\} \cup \dots \cup \{w_k\} & \text{(D.2.2)} \\
 = & \{w_1, \dots, w_k\} \\
 = & L & \square
 \end{aligned}$$

Another property which is widely known is the so called Pumping Lemma for regular languages. It describes a necessary condition for a language to be regular and was first discovered by Rabin and Scott [RS59] and reads as follows:

Remark R.2.3 (PUMPING LEMMA FOR REGULAR LANGUAGES):

For every regular language L there exists a constant $n \in \mathbb{N}$ such that for every word $w \in L$ with $|w| \geq n$ there exists a decomposition $w = uvz$, such that

- $|uv| \leq n$,

- $v \neq \varepsilon$, and
- for every $i \in \mathbb{N}$ it holds that $uv^iz \in L$.

The Pumping Lemma describes a necessary condition for a language to be regular. It is hence used for showing that a language is not regular, namely when the language in question does not respect the Pumping Lemma. The condition described in the Pumping Lemma, however, is not a sufficient condition for a language to be regular, as there exist non-regular languages which satisfy the Pumping Lemma as well.

2.1.2 Context-Free Languages

The next class in the Chomsky-Hierarchy are the context-free languages (*CFL*). The Chomsky-Hierarchy does in fact constitute a hierarchy, as $\text{Reg} \subset \text{CFL}$. The class of context-free languages, too, has been well-studied and there are hence also various characterizations of the context-free languages. A very prominent one is their characterization by means of context-free grammars. Another important characterization of the context-free languages is by means of their accepting devices — the non-deterministic pushdown automata. Since we will not be needing any details of how context-free languages are accepted or generated in this thesis, we omit recalling context-free grammars and pushdown automata.

What we will, however, need are the closure properties of the context-free languages:

Remark R.2.4 (CLOSURE PROPERTIES OF *CFL* [HMU02, Ric08]):

- (I) *CFL* is **closed** under union, concatenation, Kleene-closure operation and intersection with a regular language.
- (II) *CFL* is **not closed** under intersection (with some other context-free language) or complementation.

As we can see, the context-free languages are already much less grateful than the regular languages as far as their closure properties. Moreover, important decision problems become undecidable for context-free languages, too:

Remark R.2.5 (DECIDABILITY OF PROBLEMS FOR *CFL* [HMU02, Ric08]):

- (I) The Word Problem and the Emptiness Problem are **decidable** for *CFL*.
- (II) The Inclusion Problem and the Equivalence Problem are **undecidable** for *CFL*.

2.1.3 Context-Sensitive Languages

Next in the Chomsky-Hierarchy are the context-sensitive languages (*CSL*). It holds that $\text{CFL} \subset \text{CSL}$. For *CSL*, too, there are various equivalent characterizations, the most important ones being their definition by means of context-sensitive grammars and their definition by means of linear bounded automata. Linear bounded automata are Turing Machines, which accept or reject a word w without visiting a cell on their

working tape, which has not been occupied by a symbol of w before starting the computation. This is equivalent to using a Turing Machine which accepts or rejects a word w with space complexity $\mathcal{O}(|w|)$, i.e. the space complexity of the Turing Machine is linearly bounded by the length of the input.

In terms of their closure properties, the context-sensitive languages behave more benign than the context-free languages:

Remark R.2.6 (CLOSURE PROPERTIES OF CSL [Ric08]):

CSL is closed under union, intersection, complementation, concatenation and Kleene-closure operation.

While this seems quite desirable, the drawback is that many important decision problems become undecidable for the class of context-sensitive languages. Merely the Word Problem remains decidable:

Remark R.2.7 (DECIDABILITY OF PROBLEMS FOR CSL [Ric08]):

- (I) *The Word Problem is **decidable** for CSL .*
- (II) *The Emptiness Problem, the Inclusion Problem and the Equivalence Problem are **undecidable** for CSL .*

2.1.4 Recursively Enumerable Languages

The highest type in the Chomsky-Hierarchy, are the recursively enumerable languages (\mathcal{RE}). It holds that $CSL \subset \mathcal{RE}$. The recursively enumerable languages are one of the most studied formal languages. There is a vast variety of equivalent characterizations of \mathcal{RE} . One of the earliest is by means of Turing Machines. The closure properties of the recursively enumerable languages are given as follows:

Remark R.2.8 (CLOSURE PROPERTIES OF \mathcal{RE} [Ric08]):

- (I) *\mathcal{RE} is **closed** under union, intersection, concatenation and Kleene-closure operation.*
- (II) *\mathcal{RE} is **not closed** under complementation.*

Despite their fairly good closure properties, virtually every relevant decision problem is undecidable for the class of recursively enumerable languages:

Remark R.2.9 (DECIDABILITY OF PROBLEMS FOR \mathcal{RE} [Ric08, DSW94]):

The Word Problem, the Emptiness Problem, the Inclusion Problem and the Equivalence Problem are undecidable for \mathcal{RE} .

2.2 ω -Words and ω -Languages

The concept of ω -languages generalizes the notion of languages of words to languages of *infinite-length* words. These so called ω -words have a length of ω , where ω is the smallest infinite ordinal number, namely the cardinality of the natural numbers.

An ω -word w over the alphabet Σ is an infinite (yet countable) sequence of symbols from the alphabet Σ , e.g. $w = abababab \dots$. For infinite repetition of an ordinary word v we write v^ω . So we could rewrite $w = abababab \dots$ as $w = (ab)^\omega$. The **set of all ω -words** over the alphabet Σ is denoted by Σ^ω . For our convenience, we denote the **set of all ordinary words and all ω -words** over the alphabet Σ , i.e. the set $\Sigma^* \cup \Sigma^\omega$, by Σ^∞ .

An ω -language L is a set of ω -words $L \subseteq \Sigma^\omega$, e.g. $L = \{b^\omega, ab^\omega, aab^\omega, aaab^\omega, \dots\}$.

An ordinary word $w \in \Sigma^*$ and an ω -word $v \in \Sigma^\omega$ can be **concatenated** by the \cdot -**operator** to generate a new word, e.g. $a \cdot ab^\omega = aab^\omega$ or $bbbb \cdot b^\omega = b^\omega$. Analogously, two languages L_1 and L_2 can be concatenated, but only if L_1 contains no ω -word.

Analogously to the Kleene-star, we can generate *infinite* concatenations with the ω -**closure** or ω -**power**. For all languages $L \subseteq \Sigma^* \setminus \{\varepsilon\}$ the ω -power L^ω contains all infinite concatenations of the language L with itself. It is thus defined as

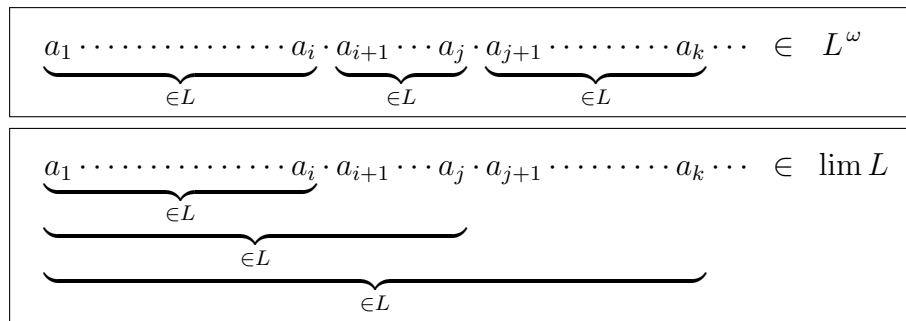
$$L^\omega = \prod_{i=1}^{\infty} L = L \cdot L \cdot L \cdot L \cdot L \dots$$

E.g. $\{a\}^\omega = \{a^\omega\}$ and $\{a, bb\}^\omega = \{w \mid w \text{ is an } \omega\text{-word over the alphabet } \{a, b\} \text{ where the number of consecutive } b\text{'s is always even}\}$.

Another concept for lifting an ordinary language into the domain of ω -languages is the **limit** of a language $L \subseteq \Sigma^* \setminus \{\varepsilon\}$, denoted by **lim** L . The limit of a language is defined as

$$\lim L = \left\{ w \in \Sigma^\omega \mid |\text{Pref}(w) \cap L| = \infty \right\},$$

where **Pref**(w) denotes the **set of prefixes of an ω -word w** . So a word w is in $\lim L$, if there exist infinitely many prefixes of w which are in the language L . E.g. $\lim\{a, b\} = \emptyset$, $\lim\{a, b\}^+ = \{a, b\}^\omega$, $\lim\{ab\}^+ = \{(ab)^\omega\}$ and $\lim L(a \cdot b^* \cdot c) = \emptyset$. To get more of a visual intuition of the meaning of the expressions L^ω and $\lim L$, consider the following figure:



2.2.1 ω -Regular Languages

We now recall the generalization of the notion of regularity for ω -languages. As for the case of ordinary regular languages, there are various characterizations of the ω -regular languages (Reg_ω), like for instance logical characterization by means of S1S formulas (Second-Order Theory of One Successor) [Büc66]. In this thesis, however, we will concentrate on the characterization by means of so called Büchi Automata and by means of ω -powers of regular languages.

2.2.1.1 Büchi Automata

The characterizations of the ω -regular languages by means of Büchi Automata is one of the best-known ones. It was developed by Julius R. Büchi for the studies of decidability of the aforementioned S1S logic. As in the case of ordinary regular languages, there are deterministic and non-deterministic variants of Büchi Automata, but unlike in the case of ordinary regular languages their expressiveness differs. We first recall the definition of the deterministic Büchi Automata:

Definition D.2.3 (DETERMINISTIC BÜCHI AUTOMATA):

A **deterministic Büchi Automaton (DBA)** \mathfrak{A} is a quintuple $\mathfrak{A} = (Q, \Sigma, \delta, q_0, F)$ where $Q, \Sigma, \delta, q_0, F$ are defined as in the definition of DFA (cf. Definition D.2.1).

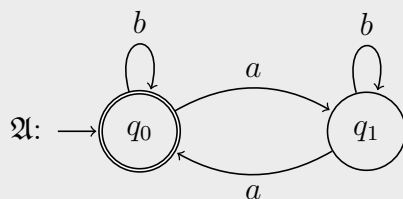
The **run ρ_w of \mathfrak{A} on a word w** is an infinite sequence of states $\rho_w \in Q^\omega$ in which $\rho_w[1] = q_0$ and in which for all $i \in \mathbb{N}$ it holds that $\delta(\rho_w[i], w[i]) = \rho_w[i+1]$. The run ρ_w of \mathfrak{A} on a word w is an **accepting run**, if it contains infinitely many final states, i.e. there exists a set of positions $M \subseteq \mathbb{N}$ such that $|M| = \infty$ and for all $i \in M$ it holds that $\rho_w[i] \in F$.

The automaton \mathfrak{A} **accepts w** , if the run of \mathfrak{A} on w is accepting, otherwise \mathfrak{A} **rejects w** . The **language $L(\mathfrak{A})$ recognized by a DBA \mathfrak{A}** is the set of all words which are accepted by \mathfrak{A} , hence $L(\mathfrak{A}) = \{w \in \Sigma^\omega \mid \mathfrak{A} \text{ accepts } w\}$.

We say that a language L is **deterministically Büchi recognizable**, if there exists a DBA \mathfrak{A} such that $L(\mathfrak{A}) = L$.

Example E.2.3 (DETERMINISTIC BÜCHI AUTOMATA):

Reconsider the DFA from Example E.2.1:



Using the automaton \mathfrak{A} as a DBA, \mathfrak{A} yields the language $\{w \in \{a, b\}^\omega \mid \text{the number of } a\text{'s in } w \text{ is either even or infinite}\}$. The run of \mathfrak{A} on the word $aababab^\omega$ is $q_0q_1q_0q_0q_1q_1q_0^\omega$. This run is accepting since the final state q_0 appears infinitely often in the run. Thus \mathfrak{A} accepts $aababab^\omega$.

The run of \mathfrak{A} on the word $aabab^\omega$ is $q_0q_1q_0q_0q_1^\omega$. This run is *not* accepting since the final state q_0 appears only 3 times in the run. Thus \mathfrak{A} rejects $aabab^\omega$.

As a final example the run of \mathfrak{A} on the word a^ω is q_0^ω which consists only of infinitely many final states and is therefore accepting and hence \mathfrak{A} accepts a^ω .

Next, we recall the definition of non-deterministic Büchi Automata which, just like NFA, use a transition relation instead of a transition function:

Definition D.2.4 (NON-DETERMINISTIC BÜCHI AUTOMATA):

A **non-deterministic Büchi Automaton (NBA)** \mathfrak{A} is a quintuple $\mathfrak{A} = (Q, \Sigma, \Delta, q_0, F)$ defined just like a DBA, except that it has a transition relation $\Delta \subseteq Q \times \Sigma \times Q$ instead of a transition function δ .

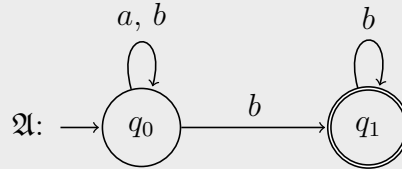
A **run ρ_w of \mathfrak{A} on a word w** is an infinite sequence of states $\rho_w \in Q^\omega$ in which $\rho_w[1] = q_0$ and in which for all $i \in \mathbb{N}$ the transition relation $\Delta(\rho_w[i], w[i], \rho_w[i+1])$ holds. A run ρ_w of \mathfrak{A} on a word w is an **accepting run**, if it contains infinitely many final states, i.e. there exists a set of positions $M \subseteq \mathbb{N}$ such that M is infinite and for all $i \in M$ it holds that $\rho_w[i] \in F$.

The automaton \mathfrak{A} **accepts w** , if there exists an accepting run of \mathfrak{A} on w , otherwise \mathfrak{A} **rejects w** . The **language $L(\mathfrak{A})$ recognized by a NBA \mathfrak{A}** is the set of all words which are accepted by \mathfrak{A} , hence $L(\mathfrak{A}) = \{w \in \Sigma^\omega \mid \mathfrak{A} \text{ accepts } w\}$.

We say that a language L is **non-deterministically Büchi recognizable** (or simply **Büchi recognizable**) or **ω -regular**, if there exists a NBA \mathfrak{A} such that $L(\mathfrak{A}) = L$.

Example E.2.4 (NON-DETERMINISTIC BÜCHI AUTOMATA):

Consider the following NBA \mathfrak{A} :



\mathfrak{A} gives rise to the language $\{w \in \{a, b\}^\omega \mid \text{From some point onwards, only } b\text{'s occur in } w\}$. A run of \mathfrak{A} on the word $aababab^\omega$ is $q_0q_0q_0q_0q_0q_0q_1^\omega$. This run is accepting since the final state q_1 appears infinitely often in the run. Thus \mathfrak{A} accepts $aababab^\omega$.

For the word $(ab)^\omega$, on the other hand, the only possible run of \mathfrak{A} is q_0^ω , because if \mathfrak{A} goes into state q_1 it cannot go back to state q_0 but processing the symbol a is only possible in state q_0 . The run q_0^ω is thus *not* accepting since the final state q_1 does not appear in the run. Thus \mathfrak{A} rejects $(ab)^\omega$.

As mentioned before, unlike the case for ordinary finite automata, the class of deterministically Büchi recognizable languages differs from the class of non-deterministically Büchi recognizable languages. Namely the deterministically Büchi recognizable languages are a *proper* subset of the non-deterministically Büchi recognizable languages [Tho90]. In fact, the language from Example E.2.4 is not deterministically Büchi recognizable.

2.2.1.2 Characterization by Means of the ω -Power

The second characterization we examine of the ω -regular languages is a characterization by means of the ω -power of ordinary regular languages. This characterization was also discovered by Büchi and represents in some way a notion of ω -regular expressions.

Remark R.2.10 (CHARACTERIZATION OF $\mathcal{R}eg_\omega$ [Büc66, Tho90]):

- (I) Every ω -regular language is a finite union of languages of the form $W \cdot V^\omega$, where W and V are regular languages.
- (II) If L_1 is a regular language and L_2 and L_3 are ω -regular languages, then $L_1 \cdot L_2$, $L_2 \cup L_3$, $L_2 \cap L_3$ and $\Sigma^\omega \setminus L_2$ are also ω -regular languages.

2.2.1.3 Closure Properties and Decision Problems

Due to Remark R.2.10 (II), we have already seen that the ω -regular languages exhibit good closure properties. Namely we have the following:

Remark R.2.11 (CLOSURE PROPERTIES OF $\mathcal{R}eg_\omega$):

$\mathcal{R}eg_\omega$ is closed under union, intersection, complementation and left-concatenation with regular languages.

Also, many decision problems are decidable for the class of ω -regular languages:

Remark R.2.12 (DECIDABILITY OF PROBLEMS FOR $\mathcal{R}eg_\omega$):

The Emptiness Problem, the Inclusion Problem and the Equivalence Problem are decidable for $\mathcal{R}eg_\omega$.

Notice that we did not consider the decidability of the Word Problem here. This is due to the fact that the Word Problem is not very meaningful in the context of ω -languages. For a short elaboration on that matter, please refer to Section 5.4.

2.2.2 ω -Context-Free Languages

Naturally, in the studies of ω -languages one has tried to adapt concepts like the Chomsky-Hierarchy from classical language theory. Therefore, the next level above ω -regular has to be the ω -context-free languages (\mathcal{CFL}_ω). Those languages are accepted by ω -pushdown automata or generated by context-free grammars which explicitly allow for infinite leftmost derivations. We will, however, not get any further into the details of accepting or generating ω -context-free languages, but we will need the following two results about their closure properties and decision problems:

Remark R.2.13 (CLOSURE PROPERTIES OF \mathcal{CFL}_ω [Sta97, CG77a, CG77b]):

- (I) \mathcal{CFL}_ω is **closed** under union, intersection with an ω -regular language and left-concatenation with context-free languages.
- (II) \mathcal{CFL}_ω is **not closed** under complementation and intersection (with some other ω -context-free language).

Remark R.2.14 (DECIDABILITY OF PROBLEMS FOR CFL_ω [CG77a, CG78a]):

- (I) *The Emptiness Problem is **decidable** for CFL_ω .*
- (II) *The Inclusion Problem and the Equivalence Problem are **not decidable** for CFL_ω .*

2.2.3 Remarks on Context-Sensitive and Recursive ω -Languages

As we mentioned in Section 2.1.3, the ordinary context-sensitive languages are accepted by Turing Machines whose space complexity is linearly bounded. One might now want to try to find some analogous concept for ω -languages, however, Ludwig Staiger remarks in [Sta97] that “because of the infinite length of the inputs, it is (...) difficult to define classes of ω -languages by means of space or time complexity.” Moreover, Staiger mentions the following result due to Rina S. Cohen and Arie Y. Gold: For every recursively enumerable language $E \subseteq \Sigma^*$ there exists a context-sensitive language $C \subseteq \Sigma^*$, such that $E \cdot \Sigma^\omega = C \cdot \Sigma^\omega$ and $\lim E = \lim C$ [CG78b]. Intuitively speaking this means that the class of recursively enumerable languages and the class of context-sensitive languages in some sense coincide when being lifted to the domain of ω -languages. Thus we can directly consider the class of ω -languages accepted by Turing Machines, which we call the **ω -recursive** languages \mathcal{RE}_ω .

The class of ω -recursive languages itself forms quite a rich hierarchy. The level of a language L in this hierarchy depends on (a) whether the Turing Machine is deterministic or non-deterministic and (b) which type of acceptance condition needs to be chosen in order for the Turing Machine to accept exactly the words in L . In his survey Staiger mentions as much as *six* different acceptance conditions for Turing Machines on ω -words, such as e.g.

- a final state must be visited at least once (1-acceptance),
- a final state must be visited infinitely often (2-acceptance),
- the set of states, that is visited infinitely often, is precisely the set of final states (3-acceptance)

However, all major decision problems are undecidable for Turing Machines on ω -words in general [Lan69], which is why we will consider \mathcal{RE}_ω as intractable in this thesis.

3

Syntax and Semantics of LOOP+ ω Programs

In this chapter we introduce LOOP+ ω , a novel model programming language for distributed reactive systems. LOOP+ ω is built on top of the well-known programming language LOOP but extends LOOP by I/O-operations and infinite loops. As LOOP is the cornerstone of LOOP+ ω , we will start this chapter off by recalling the syntax and semantics of ordinary LOOP programs before we gradually extend LOOP by message passing and infinite loops, thus obtaining the new language LOOP+ ω .

3.1 Ordinary LOOP Programs

LOOP is an imperative model programming language with the convenient characteristic that it captures exactly the primitive recursive functions. The syntax of LOOP however, is way closer to that of real programming languages than to the classical syntax of primitive recursive functions. This makes it very easy to translate real world programs into LOOP programs. Such a translation is of course more complicated when translating to primitive recursive functions. At the same time LOOP is a very minimalistic programming language with an inductive structure. This allows for easier proofs than real programming languages.

3.1.1 Syntax of LOOP Programs

All LOOP programs are finite programs which operate on an infinite yet countable set of program variables $\mathcal{Var} = \{x_1, x_2, \dots\}$. LOOP programs are constructed of different statements which modify the program variables in a certain way. In order to understand how LOOP programs operate, we first need to give a formal definition of the syntax of LOOP programs:

Definition D.3.1 (SYNTAX OF LOOP PROGRAMS):

The **syntax of LOOP programs** is defined by the following grammar:

$$\begin{aligned}
 P &\longrightarrow \mathbf{skip} \\
 &\quad | x_i := x_j + c \\
 &\quad | P; P \\
 &\quad | \mathbf{LOOP } x \mathbf{ DO } P \mathbf{ END},
 \end{aligned}$$

where $c \in \{-1, 0, 1\}$. We denote the **set of all LOOP programs** which are generated by the grammar above by \mathbb{L} .

We now explain the grammar in Definition D.3.1 in more detail: The first two rules are for single non-composed statements. **skip** is called the empty statement. $x_i := x_j + c$ is called an assignment. In assignments any two program variables x_i and x_j and any constant $c \in \{-1, 0, 1\}$ may be used.

The other two rules are for composed statements. LOOP programs can be concatenated. So if both P_1 and P_2 are LOOP programs then the concatenation $P_1; P_2$ is also a LOOP program. The second type of composed statements are loops. A loop **LOOP** x **DO** P **END** consists of a loop header and a *loop body* P . The loop body P itself must of course also be a LOOP program.

3.1.2 Semantics of LOOP Programs

Next, we recall the semantics of LOOP programs. The semantics describes the execution of a program. In particular it defines in which way program variables are modified during the course of the execution.

To represent the values assigned to the program variables a valuation function $\sigma: \mathcal{Var} \rightarrow \mathbb{N}$ is used. Such a function σ we will also call the *configuration* of a program. Even though the term configuration is usually used in the context of operational semantics, in this case we will use this term in context of denotational semantics.

We use the following notation for such valuation functions: A valuation function $\sigma_{(v_0, \dots, v_k)}$ is a function in which the i^{th} variable evaluates to v_i and all other variables x_j , for $j > k$, evaluate to 0. E.g. the valuation function $\sigma_{(2,3)}$ is given by

$$\begin{aligned}
 \sigma_{(2,3)}(x_0) &= 2 \\
 \sigma_{(2,3)}(x_1) &= 3 \\
 \sigma_{(2,3)}(x_j) &= 0, \text{ for all } j \geq 2.
 \end{aligned}$$

Using such valuation functions, we can now define the semantics of LOOP programs:

Definition D.3.2 (SEMANTICS OF LOOP PROGRAMS):

The semantics $\langle \cdot \rangle: \mathbb{L} \rightarrow (\{\mathcal{Var} \rightarrow \mathbb{N}\} \rightarrow \{\mathcal{Var} \rightarrow \mathbb{N}\})$ of a LOOP program is defined as follows:

$$\begin{aligned}
 \langle \mathbf{skip} \rangle(\sigma) &= \sigma \\
 \langle x_i := x_j + c \rangle(\sigma) &= \sigma[x_i \leftarrow \sigma(x_j) + c]
 \end{aligned}$$

$$\begin{aligned}\langle P_1; P_2 \rangle(\sigma) &= \langle P_2 \rangle(\langle P_1 \rangle(\sigma)) \\ \langle \text{LOOP } x \text{ DO } P \text{ END} \rangle(\sigma) &= \langle P \rangle^{\sigma(x)}(\sigma)\end{aligned}$$

We now describe the semantics of LOOP programs in more detail: The empty statement `skip` does not modify the program variables. It can be compared to the `NOOP` instruction of assembler programs.

The assignment $x_i := x_j + c$ modifies the value of the program variable x_i . It assigns the value of $x_j + c$ to x_i . So in the definition above $\sigma[x_i \leftarrow \sigma(x_j) + c]$ is just short for a valuation function σ' which evaluates every variable except for x_i to the same value as σ . For x_i however, σ' evaluates to $\sigma(x_j) + c$. More formally for σ' it holds that $\forall x \in \text{Var} \setminus \{x_i\}: \sigma'(x) = \sigma(x)$ and $\sigma'(x_i) = \sigma(x_j) + c$. Note that, since LOOP operates on natural numbers, we agree on $0 + (-1) = 0$.

The concatenation $P_1; P_2$ has the effect of first executing the program P_1 resulting in some intermediate configuration $\langle P_1 \rangle(\sigma) =: \sigma'$. After that, the program P_2 is executed starting with this intermediate configuration σ' . So the resulting semantics is $\langle P_2 \rangle(\sigma') = \langle P_2 \rangle(\langle P_1 \rangle(\sigma))$.

Finally, the loop `LOOP x DO P END` executes the program P a number of $\sigma(x)$ times. It is important to note that the loop body P is executed exactly as many times as the value of x was *before* executing the loop body for the first time even if the value of x is modified during the execution of the loop. So a different way to express the semantics of the loop would be

$$\langle \text{LOOP } x \text{ DO } P \text{ END} \rangle(\sigma) = \underbrace{\langle P; P; \dots; P \rangle}_{\sigma(x) \text{ times}}(\sigma).$$

Example E.3.1 (LOOP PROGRAM FOR SIMPLE ADDITION):

We finish this section by giving a small example of a program for adding the values of x_0 and x_1 . The program is given by:

```
1: LOOP  $x_0$  DO
2:    $x_1 := x_1 + 1$ 
3: END
```

It loops over x_0 and adds a 1 a number of x_0 times to x_1 . The result of the addition will be stored in x_1 after the execution terminates.

If we want to use the program above to calculate the sum of 2 and 3, we therefore simply calculate the semantics of the program starting with the valuation function $\sigma_{(2,3)}$ and then read the value of x_1 from the resulting valuation function:

$$\begin{aligned}& (\langle \text{LOOP } x_0 \text{ DO } x_1 := x_1 + 1 \text{ END} \rangle (\sigma_{(2,3)})) (x_1) \\ &= (\langle x_1 := x_1 + 1 \rangle^{\sigma_{(2,3)}(x_0)} (\sigma_{(2,3)})) (x_1) \\ &= (\langle x_1 := x_1 + 1 \rangle^2 (\sigma_{(2,3)})) (x_1) \\ &= (\langle x_1 := x_1 + 1 \rangle (\langle x_1 := x_1 + 1 \rangle (\sigma_{(2,3)}))) (x_1) \\ &= (\langle x_1 := x_1 + 1 \rangle (\sigma_{(2,3)}[x_1 \leftarrow \sigma_{(2,3)}(x_1) + 1])) (x_1)\end{aligned}$$

$$\begin{aligned}
&= (\langle x_1 := x_1 + 1 \rangle (\sigma_{(2,3)}[x_1 \leftarrow 3 + 1])) (x_1) \\
&= (\langle x_1 := x_1 + 1 \rangle (\sigma_{(2,3)}[x_1 \leftarrow 4])) (x_1) \\
&= (\langle x_1 := x_1 + 1 \rangle (\sigma_{(2,4)})) (x_1) \\
&= (\sigma_{(2,4)}[x_1 \leftarrow \sigma_{(2,4)}(x_1) + 1]) (x_1) \\
&= (\sigma_{(2,4)}[x_1 \leftarrow 4 + 1]) (x_1) \\
&= (\sigma_{(2,4)}[x_1 \leftarrow 5]) (x_1) \\
&= \sigma_{(2,5)}(x_1) \\
&= 5
\end{aligned}$$

which leaves the result of 5 as the value of variable x_1 . So in short we have $\langle \text{LOOP } x_0 \text{ DO } x_1 := x_1 + 1 \text{ END} \rangle (\sigma_{(2,3)})(x_1) = \sigma_{(2,5)}(x_1) = 5$.

In the next two sections we study extensions of the LOOP programming language which are needed for the novel model programming language LOOP+ ω . First, we introduce message passing mechanisms and later we introduce infinite loops.

3.2 Message Passing

In order to model a reactive distributed system in which the processes communicate with each other and thereby information is shared between the processes, we need to introduce some message passing mechanisms to our programming language.

Our final goal is to model a system of N communicating processes. Each process shall be able to *send* messages to a different process. In addition it is supposed to be able to *receive* messages from other processes and react to them depending on the content of the message.

For message reception every process I shall provide a FIFO queue for every other process J in which messages sent from process J to process I are queued. Such queues in this field are usually called *channels* [BZ83, Hoa78, VC83]. To avoid confusion we use the following convention:

Definition D.3.3 (NOMENCLATURE OF CHANNELS):

We denote the channel which is *located at process I* and which is designated to *queuing messages from process J* by $C_{J \rightarrow I}$.

3.2.1 Syntax of Message Passing

The first statements we add to \mathbb{L} are message sending statements:

$$\begin{aligned}
&\text{send}_J \ a \\
&\text{send}_J \ b
\end{aligned}$$

These two statements are used by process I to send either the symbol a or the symbol b to process J , or more precisely to the channel $C_{I \rightarrow J}$, i.e. the channel located at process J which is dedicated to queuing messages from process I .

For receiving messages we add another statement to \mathbb{L} :

$$\text{RECV}_J \text{ a DO } P_1 \text{ OR b DO } P_2 \text{ OR } _ \text{ DO } P_3 \text{ END}$$

With this statement, process I reads one symbol from the channel $C_{J \rightarrow I}$, i.e. the channel located at process I which is dedicated to queuing messages from process J . There are three cases for reading from the channel: Either the next symbol in the channel is an a , then process I shall react by executing program P_1 . Or the next symbol is a b , then P_2 shall be executed. The third case is that the channel is empty. In this case, the process shall react by executing program P_3 .

3.2.2 The I/O–Language

By introducing interaction with the environment through message passing, defining the semantics of programs gets more complicated than in the case of ordinary LOOP programs. On one hand, just as with ordinary LOOP programs, we need to keep track of the internal state of the program, i.e. the values of the program variables. On the other hand, the message passing statements generate I/O–events which we mainly want to keep track of in an I/O–language.

Unfortunately, both the internal state and the I/O–language of a program are linked together by the receive statement and are hence strongly interdependent. Therefore their semantics cannot be defined independently. To resolve this issue, we extend our notion of configurations to tuples, each consisting of a valuation function and a word which captures the history of I/O–events.

Furthermore, since the receive statement introduces some sort of nondeterminism, we need to keep track of *multiple* configurations at the same time, so our semantics has to operate on finite *sets* of configurations $S \subseteq \{\text{Var} \rightarrow \mathbb{N}\} \times \Sigma_{\text{I/O}}^*$.

3.2.2.1 Alphabet of the I/O–Language

In the I/O–language we want to capture every sending of a message and every reading of a message from the channels. That way, we can model both that a message was sent to and that the message was read by a process. We also want to be able to capture a reading attempt from an empty channel. In addition to the communication over the channels, we also want some sort of notion that the process is alive and proceeds with its execution.

To capture all these aspects we consider the I/O–language of a program to be over the following alphabet:

Definition D.3.4 (I/O–LANGUAGE ALPHABET):

The alphabet of the I/O–language generated by communicating programs shall be given by

$$\Sigma_{I/O} = \left\{ \langle I \xrightarrow{a} J |, \langle I \xrightarrow{b} J |, | J \xrightarrow{a} I \rangle, \right. \\ \left. | J \xrightarrow{b} I \rangle, | J \xrightarrow{\perp} I \rangle, \sim^I \sim \mid I, J \in \{1, \dots, N\} \right\}.$$

The symbol $\langle I \xrightarrow{a} J |$ represents the event of process I sending the symbol a to channel $C_{I \rightarrow J}$. The symbol $| J \xrightarrow{a} I \rangle$ represents the event of process I reading the symbol a from channel $C_{J \rightarrow I}$. Explanations for $\langle I \xrightarrow{b} J |$ and $| I \xrightarrow{b} J \rangle$ are analogous. The symbol $| J \xrightarrow{\perp} I \rangle$ represents the event of process I unsuccessfully attempting to read from channel $C_{J \rightarrow I}$, i.e. I attempts to read from an empty channel.

The symbol $\sim^I \sim$ symbolizes that the process I proceeds with its execution. This is needed much later for ensuring some sort of fairness in the system of communicating programs and to make sure that the language generated by a program is in fact a language containing *only* infinite words. At this point suffice it to say that the symbol $\sim^I \sim$ will represent the completion of one iteration of an infinite loop. We will introduce those infinite loops in Section 3.3 and the interplay of infinite loops with the $\sim^I \sim$ symbol in Section 3.4, but for the sake of introducing the alphabet $\Sigma_{I/O}$ as a whole we introduced the symbol already at this point.

3.2.2.2 Concatenations of Words and Configurations

In order to make the upcoming definitions and proofs more legible, we introduce a concatenation of words and configurations. The concatenation shall be defined in a way such that it appends a common prefix (or suffix) w to all words in the I/O-language which is represented by a set of configurations. The concatenation is defined as follows:

Definition D.3.5 (CONCATENATION OF WORDS AND CONFIGURATIONS):

Let w be an I/O-word and S be a set of configurations. Then the concatenation $w \cdot S$ is defined as

$$w \cdot S = \{(\sigma, w \cdot v) \mid (\sigma, v) \in S\}$$

and the concatenation $S \cdot w$ is defined analogously as

$$S \cdot w = \{(\sigma, v \cdot w) \mid (\sigma, v) \in S\}.$$

So $w \cdot S$ means adding a common prefix and $S \cdot w$ means adding a common suffix w to all the I/O-words in S . Note that by introducing infinite ω -loops later, the words in the configurations will become infinite, so adding a common suffix is of course only reasonable for sets of configurations which contain only finite words, i.e. sets of configurations that emerge from executing infinite-loop-free programs.

As we can easily observe the concatenation as defined above is associative. This follows almost directly from Definition D.3.5 and from the fact, that the ordinary concatenation of words is associative.

Corollary C.3.1 (ASSOCIATIVITY OF THE CONCATENATION):

The concatenation of prefixes and suffixes is associative, i.e. for any words $w \in \Sigma_{I/O}^*$ and $z \in \Sigma_{I/O}^\infty$ and any set of configurations S with only finite words the following equality holds:

$$(w \cdot S) \cdot z = w \cdot (S \cdot z)$$

Proof:

$$\begin{aligned} & (w \cdot S) \cdot z \\ &= \{(\sigma, w \cdot v) \mid (\sigma, v) \in S\} \cdot z && \text{(D.3.5)} \\ &= \{(\sigma, (w \cdot v) \cdot z) \mid (\sigma, v) \in S\} && \text{(D.3.5)} \\ &= \{(\sigma, w \cdot (v \cdot z)) \mid (\sigma, v) \in S\} && \text{(concat. of words is associat.)} \\ &= w \cdot \{(\sigma, (v \cdot z)) \mid (\sigma, v) \in S\} && \text{(D.3.5)} \\ &= w \cdot (S \cdot z) && \text{(D.3.5)} \end{aligned}$$

□

3.2.3 Semantics of Message Passing

In the following, we define complete semantics for message passing LOOP programs and explain those in detail afterwards.

Definition D.3.6 (SEMANTICS OF LOOP WITH MESSAGE PASSING):

The semantics

$$\llbracket \cdot \rrbracket : \mathbb{L} \rightarrow \left(\mathcal{P}(\{\mathcal{V}ar \rightarrow \mathbb{N}\} \times \Sigma_{I/O}^*) \rightarrow \mathcal{P}(\{\mathcal{V}ar \rightarrow \mathbb{N}\} \times \Sigma_{I/O}^*) \right)$$

of LOOP programs with message passing is given as follows:

$$\begin{aligned} \llbracket \text{skip} \rrbracket(S) &= S \\ \llbracket x_i := x_j + c \rrbracket(S) &= \{(\sigma[x_i \leftarrow \sigma(x_j) + c], w) \mid (\sigma, w) \in S\} \\ \llbracket \text{send}_J \ a \rrbracket(S) &= S \cdot \mathcal{I} \xrightarrow{a} J \\ \llbracket \text{send}_J \ b \rrbracket(S) &= S \cdot \mathcal{I} \xrightarrow{b} J \\ \llbracket \text{RECV}_J \ a \ \text{DO} \ P_1 \\ &\quad \text{OR} \ b \ \text{DO} \ P_2 \\ &\quad \text{OR} \ _ \ \text{DO} \ P_3 \ \text{END} \rrbracket(S) &= \llbracket P_1 \rrbracket(S \cdot \mathcal{I} \xrightarrow{a} J) \cup \llbracket P_2 \rrbracket(S \cdot \mathcal{I} \xrightarrow{b} J) \\ &\quad \cup \llbracket P_3 \rrbracket(S \cdot \mathcal{I} \xrightarrow{\perp} J) \\ \llbracket P_1 ; P_2 \rrbracket(S) &= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(S)) \\ \llbracket \text{LOOP} \ x \ \text{DO} \ P \ \text{END} \rrbracket(S) &= \bigcup_{(\sigma, w) \in S} \llbracket P \rrbracket^{\sigma(x)}(\{(\sigma, w)\}) \end{aligned}$$

Again, the empty statement **skip** does not change the configuration. Neither communication nor variable modification takes place.

The assignment $x_i := x_j + c$ changes the valuation of x_i for every valuation function σ of every tuple $(\sigma, w) \in S$ to $\sigma(x_j) + c$. The I/O-words are not touched as no communication takes place.

The message sending statement $\text{send}_J \ a$ appends the symbol $\langle I \xrightarrow{a} J \mid$ to every I/O-word of every tuple in S . The valuation functions are not touched as the program variables are not modified. The case for $\text{send}_J \ b$ is analogous.

The reception statement $\text{RECV}_J \ a \ \text{DO } P_1 \ \text{OR } b \ \text{DO } P_2 \ \text{OR } _ \ \text{DO } P_3 \ \text{END}$ is a composed statement and its semantics is more complicated: This statement reads the next symbol from channel $C_{J \rightarrow I}$. Depending on which symbol was read, the further execution of the program takes a different course:

If e.g. a was the next symbol in the channel, the program P_1 would be executed. For that, we need to proceed with the semantics of P_1 on a set of configurations, where to each I/O-word the symbol $\mid J \xrightarrow{a} I \rangle$ was appended. The valuation functions are not touched by this statement, however, the semantics of P_1 may of course have some effect on them. The case for reading b from the channel is analogous: $\mid J \xrightarrow{b} I \rangle$ is appended and P_2 is executed. The third case is the reading attempt from an empty channel. In this case the execution proceeds with program P_3 and the symbol $\mid J \xrightarrow{_} I \rangle$ is appended to the I/O-words.

Since potentially every one of these three cases (reading a , reading b , reading attempt from empty channel) could have occurred, we need the semantics of the receive statement to be the union of the semantics of all three cases.

The semantics of the concatenation is straightforward: It simply concatenates the semantics of P_1 and P_2 . The loop $\text{LOOP } x \ \text{DO } P \ \text{END}$ concatenates the semantics of P a number of $\sigma(x)$ times. However, in S there might be configurations with differing valuations for x . As there are only finitely many configurations in S , we can resolve this issue by simply calculating $\llbracket P \rrbracket^{\sigma(x)}(\{(\sigma, w)\})$ for every possible configuration $(\sigma, w) \in S$ and then taking the union over those results. This way, the correct number of loop iterations is ensured for each configuration.

Example E.3.2 (LOOP PROGRAM WITH MESSAGE PASSING):

We finish this section by giving a small example of a program P_I which echoes one symbol it reads from channel $C_{J \rightarrow I}$ to channel $C_{I \rightarrow J}$. The program is given by:

```

1:  RECVJ
2:      a DO sendJ a
3:      OR b DO sendJ b
4:      OR _ DO skip
5:  END

```

The program reads a symbol from the channel. If an a is read an a is echoed and if a b is read a b is echoed, respectively. If the channel is empty, nothing is echoed.

We now calculate the semantics of the program starting with $\{(\sigma_0, \varepsilon)\}$:

$$\llbracket \text{RECV}_J \ a \ \text{DO } \text{send}_J \ a \ \text{OR } b \ \text{DO } \text{send}_J \ b \ \text{OR } _ \ \text{DO } \text{skip} \ \text{END} \rrbracket(\{(\sigma_0, \varepsilon)\})$$

$$\begin{aligned}
&= \llbracket \text{send}_J \ a \rrbracket \left(\{(\sigma_0, \varepsilon)\} \cdot |J \xrightarrow{a} I^\circ\} \right) \\
&\quad \cup \llbracket \text{send}_J \ b \rrbracket \left(\{(\sigma_0, \varepsilon)\} \cdot |J \xrightarrow{b} I^\circ\} \right) \\
&\quad \cup \llbracket \text{skip} \rrbracket \left(\{(\sigma_0, \varepsilon)\} \cdot |J \xrightarrow{\perp} I^\circ\} \right) \\
&= \left(\{(\sigma_0, \varepsilon)\} \cdot |J \xrightarrow{a} I^\circ\} \cdot \langle I \xrightarrow{a} J | \rangle \right) \\
&\quad \cup \left(\{(\sigma_0, \varepsilon)\} \cdot |J \xrightarrow{b} I^\circ\} \cdot \langle I \xrightarrow{b} a | \rangle \right) \\
&\quad \cup \left(\{(\sigma_0, \varepsilon)\} \cdot |J \xrightarrow{\perp} I^\circ\} \right) \\
&= \left\{ \left(\sigma_0, |J \xrightarrow{a} I^\circ\} \cdot \langle I \xrightarrow{a} J | \rangle \right) \right\} \\
&\quad \cup \left\{ \left(\sigma_0, |J \xrightarrow{b} I^\circ\} \cdot \langle I \xrightarrow{b} J | \rangle \right) \right\} \\
&\quad \cup \left\{ \left(\sigma_0, |J \xrightarrow{\perp} I^\circ\} \right) \right\} \\
&= \left\{ \left(\sigma_0, |J \xrightarrow{a} I^\circ\} \cdot \langle I \xrightarrow{a} J | \rangle \right), \left(\sigma_0, |J \xrightarrow{b} I^\circ\} \cdot \langle I \xrightarrow{b} J | \rangle \right), \left(\sigma_0, |J \xrightarrow{\perp} I^\circ\} \right) \right\}
\end{aligned}$$

We will study, how to interpret such semantics like the one above as a language over $\Sigma_{I/O}$ in Section 3.4.3.

3.3 ω -Loops

Reactive distributed systems are often designed in a way so that they are composed of some initialization code followed by some infinite loop. Usually one iteration of the infinite loop will poll from the incoming-channels, process the information received and then continue to the next iteration. In order to model this behavior we introduce infinite loop statements called ω -loops.

3.3.1 Syntax of ω -Loops

The syntax of ω -loops is borrowed from the one of conventional loops in ordinary LOOP programs. An ω -loop has the following syntax:

$$\text{LOOP } \omega \text{ DO } P_{\text{body}}; \text{ pulse END}$$

In an ω -loop, P_{body} must be an *ordinary* LOOP program. It must *not* contain ω -loops, as we assume only one outer infinite loop. The loop body consists of the program P_{body} and the *pulse* statement which indicates the completion of one iteration of the infinite loop.

3.3.2 Semantics of ω -Loops

From an operational point of view, an ω -loop shall behave similar to a regular loop, except that the loop body of the ω -loop is repeated infinitely many times.

From this angle we can see the conceptual difference between ω -loops and loops in LOOP or WHILE: In LOOP a loop LOOP x DO P END always terminates after a finite number of x loop iterations. A WHILE-loop WHILE $x \neq 0$ DO P END may or may not terminate, depending on whether the value of x is 0 before the next loop iteration or not. The ω -loop on the other hand will *never* terminate. The loop body is repeated infinitely many times.

For this reason, it may be that the values of the variables and therefore the valuation functions change infinitely often. In these cases it makes no sense for the corresponding tuple to contain a concrete valuation function. Thus we expand the domain of the valuation functions to $\{\mathcal{V}ar \rightarrow \mathbb{N}\} \cup \{\perp\}$, where \perp indicates non-termination and therefore potentially perpetual change of the values of the program variables.

In addition, the I/O-words resulting from the execution of an ω -loop will become infinite, as the pulse statement in the ω -loop forces *at least* the symbol $\sim\sim$ to be added to the I/O-word in each iteration of the ω -loop.

The definition of the pulse statement's semantics is straightforward:

$$\llbracket \text{pulse} \rrbracket(S) = S \cdot \sim\sim$$

The $\sim\sim$ symbol is simply added to each of the I/O-words of every tuple in S . The semantics of the ω -loop is a little more complicated:

$$\llbracket \text{LOOP } \omega \text{ DO } P \text{ END} \rrbracket(S) = \left\{ (\perp, w) \left| \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in \llbracket P \rrbracket^k(S) \end{array} \right. \right\}$$

Recall that the program P is actually of the form P_{body} ; pulse, and therefore in each iteration of the ω -loop, at least one symbol is appended.

As we can see, the valuation functions are set to \perp which represents the (potential) perpetual change of the values of the program variables. The resulting I/O-words are ω -words over the I/O-alphabet $\Sigma_{I/O}$. Every ω -word w in the set at hand is of the form $w_0 \cdot w_1 \cdot w_2 \cdots$, i.e. there exists a splitting of w into infinitely many finite words $w_i \in \Sigma_{I/O}^*$, such that for any number of ω -loop iterations k the word $w_0 \cdot w_1 \cdots w_k$ is in the semantics of k times iterating P . Note that it must hold that $w_0 \in \llbracket P \rrbracket^0(S) = S$. That means that the prefix w_0 represents a word which has been in S before entering the loop body of the ω -loop for the first time.

3.4 LOOP+ ω Programs

We have now studied all concepts required for the novel programming language LOOP+ ω . In this section, we summarize those concepts by providing the complete syntax and semantics of LOOP+ ω programs.

3.4.1 Syntax of LOOP+ ω Programs

As mentioned before, a typical program for a reactive system will consist of some initialization code followed by one outer infinite loop. To empathize this concept, we define the following syntax for LOOP+ ω programs:

Definition D.3.7 (SYNTAX OF LOOP+ ω PROGRAMS):

The syntax of a LOOP+ ω program is given by the following grammar:

$$\begin{aligned}
P_\omega &\longrightarrow P; \text{ LOOP } \omega \text{ DO } P; \text{ pulse END} \\
P &\longrightarrow \text{skip} \\
&\quad | x_i := x_j + c \\
&\quad | \text{send}_J a \\
&\quad | \text{RECV}_J a \text{ DO } P_1 \text{ OR } b \text{ DO } P_2 \text{ OR } _ \text{ DO } P_3 \text{ END} \\
&\quad | P; P \\
&\quad | \text{LOOP } x \text{ DO } P \text{ END}
\end{aligned}$$

We denote the set of all LOOP+ ω programs which are generated by the grammar above by \mathbb{L}_ω .

So every LOOP+ ω program consists of an initial LOOP program which is followed by an infinite ω -loop with another LOOP program plus the pulse as the loop body of the ω -loop. Note that the ω -loop may not be nested into another LOOP program.

3.4.2 Semantics of LOOP+ ω Programs

Incorporating the concepts from Section 3.2 and Section 3.3, we can now give the complete semantics of LOOP+ ω programs:

Definition D.3.8 (SEMANTICS OF LOOP+ ω PROGRAMS):

We denote the set of all configurations which can emerge from executing a LOOP program with message passing (thus ω -loop-free program) by

$$\mathbb{S} = \{\mathcal{V}ar \rightarrow \mathbb{N}\} \times \Sigma_{I/O}^*$$

and we denote the set of all configurations which can emerge from executing a LOOP or LOOP+ ω program by

$$\mathbb{S}_\perp = \mathbb{S} \cup \{(\mathcal{V}ar \cup \{\perp\}) \rightarrow \mathbb{N}\} \times \Sigma_{I/O}^\omega.$$

The semantics $\llbracket \cdot \rrbracket: (\mathbb{L} \cup \mathbb{L}_\omega) \rightarrow (\mathcal{P}(\mathbb{S}) \rightarrow \mathcal{P}(\mathbb{S}_\perp))$ of LOOP and LOOP+ ω programs is given as follows:

$$\begin{aligned}
\llbracket \text{skip} \rrbracket(S) &= S \\
\llbracket x_i := x_j + c \rrbracket(S) &= \{(\sigma[x_i \leftarrow \sigma(x_j) + c], w) \mid (\sigma, w) \in S\} \\
\llbracket \text{send}_J a \rrbracket(S) &= S \cdot \langle I \xrightarrow{a} J \rangle \\
\llbracket \text{send}_J b \rrbracket(S) &= S \cdot \langle I \xrightarrow{b} J \rangle \\
\llbracket \text{pulse} \rrbracket(S) &= S \cdot \sim\sim\sim \\
\llbracket \text{RECV}_J a \text{ DO } P_1 \\
&\quad \text{OR } b \text{ DO } P_2 \\
&\quad \text{OR } _ \text{ DO } P_3 \text{ END} \rrbracket(S) &= \llbracket P_1 \rrbracket(S \cdot \langle J \xrightarrow{a} I \rangle) \cup \llbracket P_2 \rrbracket(S \cdot \langle J \xrightarrow{b} I \rangle) \\
&\quad \cup \llbracket P_3 \rrbracket(S \cdot \langle J \xrightarrow{\perp} I \rangle)
\end{aligned}$$

$$\begin{aligned} \llbracket P_1; P_2 \rrbracket(S) &= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(S)) \\ \llbracket \text{LOOP } x \text{ DO } P \text{ END} \rrbracket(S) &= \bigcup_{(\sigma, w) \in S} \llbracket P \rrbracket^{\sigma(x)}(\{(\sigma, w)\}) \\ \llbracket \text{LOOP } \omega \text{ DO } P \text{ END} \rrbracket(S) &= \left\{ (\perp, w) \left| \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in \llbracket P \rrbracket^k(S) \end{array} \right. \right\} \end{aligned}$$

3.4.3 I/O–Languages of LOOP+ ω Programs

Since we are only interested in the communication behavior of the programs and since the internal values of the variables could just be put out by the program (hence all internal information on the program variables could also be coded into the I/O–language at any point of the course of the execution), we will use the following two definitions for defining the language of a LOOP+ ω program:

Definition D.3.9 (LANGUAGE OF A SET OF CONFIGURATIONS):

The language $L(S) \subseteq \Sigma_{I/O}^\infty$ which is represented by a set of configurations $S \subseteq \mathbb{S}_\perp$ is defined by

$$L(S) = \{w \mid (\sigma, w) \in S\}.$$

Definition D.3.10 (LANGUAGE OF A LOOP+ ω PROGRAM):

The language $L_S(P) \subseteq \Sigma_{I/O}^\infty$ which is generated by a program P depending on the initial configuration S is defined by

$$L_S(P) = L(\llbracket P \rrbracket(S)) = \{w \mid (\sigma, w) \in \llbracket P \rrbracket(S)\}$$

If no S is given, we assume $S = S_0 := \{(\sigma_0, \varepsilon)\}$, where σ_0 is the initial valuation function for which $\forall x_i \in \mathcal{V}ar: \sigma_0(x_i) = 0$ holds, and define

$$L(P) = L_{S_0}(P).$$

We say that a language L is **LOOP+ ω –generatable**, if there exists some LOOP+ ω program P such that $L(P) = L$. The **class of all LOOP+ ω –generated languages** we denote by $\mathcal{L}oop_\omega$.

As it will turn out, $\mathcal{L}oop_\omega$ is far beyond ω –regular or even ω –context–free. As we will discuss in Chapter 5, the class of LOOP+ ω –generated languages in fact agrees in some sense with the class of the recursively enumerable languages. Thus we will provide an over–approximation of LOOP+ ω –generated languages which guarantees ω –regularity and is therefore much more tractable in Chapter 6. We finish this section by giving an example of a LOOP+ ω program and its language:

Example E.3.3 (LOOP+ ω PROGRAM WITH INFINITE OUTPUT):

The following LOOP+ ω program is the program of process I . It sends an infinite stream of symbols a to channel $C_{I \rightarrow J}$. The program is given by:

```

1: skip;
2: LOOP  $\omega$  DO
3:     sendJ a;
4:     pulse
5: END

```

We now calculate the language of the program. For that, we first calculate the semantics of one iteration of the loop body of the ω -loop with respect to some starting configuration S_0 :

$$\begin{aligned}
& \llbracket \text{send}_J \text{ a; pulse} \rrbracket(S_0) \\
&= \llbracket \text{pulse} \rrbracket (\llbracket \text{send}_J \text{ a} \rrbracket (S_0)) \\
&= \llbracket \text{pulse} \rrbracket (S_0 \cdot \wr I \xrightarrow{a} J |) \\
&= S_0 \cdot \wr I \xrightarrow{a} J | \cdot \sim \sim \sim^I
\end{aligned}$$

If we calculated the semantics of two iterations of the ω -loop body, we would obtain $S_0 \cdot \wr I \xrightarrow{a} J | \cdot \sim \sim \sim^I \cdot \wr I \xrightarrow{a} J | \cdot \sim \sim \sim^I$. It is easy to see that for the general case of n ω -loop-body-iterations, the semantics are given by $S_0 \cdot \left(\wr I \xrightarrow{a} J | \cdot \sim \sim \sim^I \right)^n$.

We now calculate the semantics of the entire ω -loop. Recall that the semantics of an ω -loop is given as

$$\llbracket \text{LOOP } \omega \text{ DO } P \text{ END} \rrbracket(S) = \left\{ (\perp, w) \mid \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in \llbracket P \rrbracket^k(S) \end{array} \right\}.$$

So in case of the program given above, the semantics of the entire ω -loop results to

$$\left\{ (\perp, w) \mid \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in S_0 \cdot \left(\wr I \xrightarrow{a} J | \cdot \sim \sim \sim^I \right)^k \end{array} \right\}$$

which can be simplified to

$$\left\{ \left(\perp, \left(\wr I \xrightarrow{a} J | \cdot \sim \sim \sim^I \right)^\omega \right) \right\}.$$

For calculating the language of the program, we simply strip away the valuation functions (or rather the \perp which stands for the undefined valuation) and thus obtain

$$L(P) = \left\{ \left(\wr I \xrightarrow{a} J | \cdot \sim \sim \sim^I \right)^\omega \right\}$$

as the resulting language generated by the program above. In this example, the resulting language is obviously ω -regular, however, as we will see later, the languages generated by LOOP+ ω programs are generally not always ω -regular.

3.5 Syntactic Sugar

Syntactic sugar are additional syntactic rules for programming languages, which do not increase their expressiveness. In other words, syntactic sugar provides merely syntactic abbreviations. The syntax of any modern programming language consists mostly of syntactic sugar, considering that a minimalistic language such as WHILE is already Turing-complete, which means that any function that could be implemented e.g. in Java or C++ could also be implemented in WHILE. Syntactic sugar is added mainly due to the fact that the development of a program in Java takes much less time and the result is by far more maintainable than an equivalent program in WHILE or GOTO.

When employing formal methods, it is sometimes necessary to conduct proofs over the set of all possible programs. In order to keep such proofs simple, it is, however, very convenient if a language does *not* provide syntactic sugar. On the other hand, if one wants to prove that a certain program exists by providing that very program, it is sometimes convenient to have a language which provides some syntactic abbreviations. In the following we shall provide some syntactic sugar for LOOP programs which we shall use in the remainder of this thesis. We also show that these abbreviations are indeed LOOP-computable and thus do not increase the expressiveness of LOOP.

Constant Assignment

The assignment of a constant $x_i := c$ assigns a constant c to a program variable x_i . Constant assignment is LOOP-computable as it is short for

$$\begin{array}{l} \ell: \quad x_i := x_k + 0 \\ \ell + 1: \quad x_i := x_i + 1 \\ \quad \quad \vdots \\ \quad \quad \quad c \text{ times} \\ \quad \quad \quad \vdots \\ \ell + c: \quad x_i := x_i + 1, \end{array}$$

where x_k is a program variable which is not used anywhere else in the program (thus $\sigma(x_k) = 0$ for any σ).

Conditional Branching

The conditional branch IF $x_i \neq 0$ THEN P_1 ELSE P_2 END executes the program P_1 , if the valuation of x_i is unequal to 0 and otherwise executes P_2 . Conditional branching is LOOP-computable as it is short for

$$\begin{array}{l} \ell: \quad \text{LOOP } x_i \text{ DO } x_{k_1} := x_k + 1 \text{ END} \\ \ell + 1: \quad x_{k_2} := x_k + 1 \\ \ell + 2: \quad \text{LOOP } x_i \text{ DO } x_{k_2} := x_k + 0 \text{ END} \\ \ell + 3: \quad \text{LOOP } x_{k_1} \text{ DO } P_1 \text{ END} \\ \ell + 4: \quad \text{LOOP } x_{k_2} \text{ DO } P_2 \text{ END,} \end{array}$$

where x_k , x_{k_1} and x_{k_2} are program variables which are not used anywhere else in the program (thus $\sigma(x_k) = \sigma(x_{k_1}) = \sigma(x_{k_2}) = 0$ for any σ).

4

Lemmas and Corollaries on Configurations and Languages

In this chapter we discuss some properties of the algebraic structures $(\mathcal{P}(\mathbb{S}), \cup)$ and $(\mathcal{P}(\mathbb{S}_\perp), \cup)$. In particular we present some homo- and endomorphisms on these structures which will aid us in upcoming proofs. Furthermore, we discuss some properties of the LOOP- and the LOOP+ ω -generated languages.

First off, we classify the structures $(\mathcal{P}(\mathbb{S}), \cup)$ and $(\mathcal{P}(\mathbb{S}_\perp), \cup)$ in terms of what algebraic structure they form. We will show that they both form a monoid, but not a group. For that, we recall what monoids and groups are:

Definition D.4.1 (MONOIDS, SUBMONOIDS, GROUPS AND SUBGROUPS):

An algebraic structure (M, \circ, e) with a universe M , a binary operation $\circ: M \times M \rightarrow M$ and a neutral element $e \in M$ is called a **monoid**, if

- (i) \circ is a total,
- (ii) \circ is associative, i.e. for all elements $m_1, m_2, m_3 \in M$ it holds that

$$(m_1 \circ m_2) \circ m_3 = m_1 \circ (m_2 \circ m_3),$$

- (iii) $e \in M$ is a neutral element with respect to \circ , i.e. for all elements $m \in M$ it holds that

$$e \circ m = m \circ e = m.$$

The structure (M, \circ, e) is called a **group**, if it is a monoid and in addition it holds that

- (iv) for all elements $m \in M$ there exists an inverse element $m^{-1} \in M$. m^{-1} is the inverse of m if it holds that

$$m \circ m^{-1} = m^{-1} \circ m = e.$$

Furthermore a monoid or group is called **commutative**, if

(v) \circ is commutative, i.e. for all elements $m_1, m_2 \in M$ it holds that

$$m_1 \circ m_2 = m_2 \circ m_1.$$

A monoid (M', \circ, e) is called a **submonoid** of (M, \circ, e) , if $M' \subseteq M$ and (M', \circ, e) is a monoid. Analogously a group (G', \circ, e) is called a **subgroup** of (G, \circ, e) , if $G' \subseteq G$ and (G', \circ, e) is a group.

If the neutral element e of a monoid or a group (M, \circ, e) is evident, we simply speak of the monoid or group (M, \circ) . If in addition to that the operation \circ is also evident, we simply speak of the monoid or group M .

The first thing we observe about $(\mathcal{P}(\mathbb{S}), \cup)$ and $(\mathcal{P}(\mathbb{S}_\perp), \cup)$ is that both form a commutative monoid:

Corollary C.4.2:

(I) $(\mathcal{P}(\mathbb{S}_\perp), \cup, \emptyset)$ is a commutative monoid.

(II) $(\mathcal{P}(\mathbb{S}), \cup, \emptyset)$ is a commutative submonoid of $(\mathcal{P}(\mathbb{S}_\perp), \cup, \emptyset)$.

Proof of (I): For any set M , the structure $(\mathcal{P}(M), \cup, \emptyset)$ is a commutative monoid, since

- (i) \cup is always total on a powerset domain,
- (ii) \cup is always associative,
- (iii) \emptyset is element of any powerset and neutral with respect to \cup , and
- (iv) \cup is always commutative.

If for any set M , the structure $(\mathcal{P}(M), \cup, \emptyset)$ is a commutative monoid, then in particular the structure $(\mathcal{P}(\mathbb{S}_\perp), \cup, \emptyset)$ is a commutative monoid as well.

Proof of (II): $\mathbb{S} \subseteq \mathbb{S}_\perp$ and furthermore $(\mathcal{P}(\mathbb{S}), \cup, \emptyset)$ is a commutative monoid (cf. Proof of (I)). \square

Note that $(\mathcal{P}(\mathbb{S}), \cup)$ and $(\mathcal{P}(\mathbb{S}_\perp), \cup)$ do not form a group. A group would require that for all elements $S \in \mathbb{S}_\perp$ ($S \in \mathbb{S}$ respectively) there exists an element $S^{-1} \in \mathbb{S}_\perp$ ($S^{-1} \in \mathbb{S}$ respectively) such that $S \cup S^{-1} = \emptyset$ which is obviously not given: Merely the element \emptyset is self-inverse as $\emptyset \cup \emptyset = \emptyset$. All other elements do not have an inverse.

In the following, we want show that certain operations are homo- or endomorphisms on the monoids $(\mathcal{P}(\mathbb{S}), \cup)$ and $(\mathcal{P}(\mathbb{S}_\perp), \cup)$. Thus, we recall the definition of homo- and endomorphisms.

Definition D.4.2 (HOMOMORPHISMS AND ENDOMORPHISMS):

For two algebraic structures (M, \circ) and (K, \star) a function $h: M \rightarrow K$ is called a **homomorphism from (M, \circ) to (K, \star)** , if for all elements $m_1, m_2 \in M$ it holds that

$$h(m_1 \circ m_2) = h(m_1) \star h(m_2).$$

A homomorphism from (M, \circ) to (M, \circ) is called an **endomorphism on (M, \circ)** .

4.1 Endomorphic Property of the Concatenation of Words and Configurations

The first operation we study in terms of its homo- or rather endomorphic property is the concatenation of words and configurations as in Definition D.3.5: We observe that this concatenation is distributive over \cup which means that concatenation of a suffix or a prefix forms an endomorphism on the monoids $(\mathcal{P}(\mathbb{S}), \cup, \emptyset)$ and $(\mathcal{P}(\mathbb{S}_\perp), \cup, \emptyset)$, respectively. More precisely:

Corollary C.4.3 (ENDOMORPHIC PROPERTY OF CONCATENATION):

- (I) For any finite word $w \in \Sigma_{I/O}^*$ the concatenation of the common prefix w to a set of configurations is an endomorphism on $(\mathcal{P}(\mathbb{S}_\perp), \cup)$, i.e. for any two configurations $S_1, S_2 \subseteq \mathbb{S}_\perp$ the following holds:

$$(w \cdot S_1) \cup (w \cdot S_2) = w \cdot (S_1 \cup S_2)$$

- (II) For any word $w' \in \Sigma_{I/O}^\infty$ the concatenation of the common suffix w' to a set of configurations with only finite words is an endomorphism on $(\mathcal{P}(\mathbb{S}), \cup)$, i.e. for any two configurations $S'_1, S'_2 \subseteq \mathbb{S}$ the following holds:

$$(S'_1 \cdot w') \cup (S'_2 \cdot w') = (S'_1 \cup S'_2) \cdot w'$$

Proof: We only give the proof for (I):

$$\begin{aligned} & (w \cdot S_1) \cup (w \cdot S_2) \\ &= \{(\sigma, w \cdot v) \mid (\sigma, v) \in S_1\} \cup \{(\sigma, w \cdot v) \mid (\sigma, v) \in S_2\} \end{aligned} \tag{D.3.5}$$

$$\begin{aligned} &= \{(\sigma, w \cdot v) \mid (\sigma, v) \in (S_1 \cup S_2)\} \\ &= w \cdot (S_1 \cup S_2) \end{aligned} \tag{D.3.5}$$

The proof for (II) is completely analogous. \square

4.2 Homomorphic Property of the Application of LOOP or LOOP+ ω Semantics

The second operation we study in terms of its homo- or endomorphic property is the application of the semantics of a program to a set of configurations. We state that the semantics of any LOOP or LOOP+ ω program form a homomorphic mapping from $(\mathcal{P}(\mathbb{S}), \cup)$ to $(\mathcal{P}(\mathbb{S}_\perp), \cup)$.

Lemma L.4.1 (HOMOMORPHIC PROPERTY OF LOOP AND LOOP+ ω SEMANTICS):

For any LOOP+ ω or LOOP program $P \in \mathbb{L} \cup \mathbb{L}_\omega$ the semantics of the program $\llbracket P \rrbracket: \mathcal{P}(\mathbb{S}) \rightarrow \mathcal{P}(\mathbb{S}_\perp)$ is a homomorphism from (\mathbb{S}, \cup) to (\mathbb{S}_\perp, \cup) , i.e. for any two sets of configurations $S_1, S_2 \subseteq \mathbb{S}$, the following equality holds:

$$\llbracket P \rrbracket(S_1) \cup \llbracket P \rrbracket(S_2) = \llbracket P \rrbracket(S_1 \cup S_2)$$

Proof: The proof of Lemma L.4.1 is done by structural induction over all LOOP and LOOP+ ω programs. For the induction basis, we have the non-composed statements. As the induction hypothesis we assume that Lemma L.4.1 holds for programs P_1 , P_2 and P_3 . We then show that, given the induction hypothesis, Lemma L.4.1 also holds for the statements composed of P_1 , P_2 and P_3 .

Induction Basis

For the induction basis we prove Lemma L.4.1 for every non-composed statement:

Empty Statement

$$\begin{aligned} \llbracket \text{skip} \rrbracket(S_1) \cup \llbracket \text{skip} \rrbracket(S_2) &= S_1 \cup S_2 && \text{(D.3.8)} \\ &= \llbracket \text{skip} \rrbracket(S_1 \cup S_2) && \text{(D.3.8)} \end{aligned}$$

Assignment

$$\begin{aligned} &\llbracket x_i := x_j + c \rrbracket(S_1) \cup \llbracket x_i := x_j + c \rrbracket(S_2) \\ &= \{(\sigma[x_i \leftarrow \sigma(x_j) + c], w) \mid (\sigma, w) \in S_1\} && \text{(D.3.8)} \\ &\quad \cup \{(\sigma[x_i \leftarrow \sigma(x_j) + c], w) \mid (\sigma, w) \in S_2\} \\ &= \{(\sigma[x_i \leftarrow \sigma(x_j) + c], w) \mid (\sigma, w) \in S_1 \cup S_2\} \\ &= \llbracket x_i := x_j + c \rrbracket(S_1 \cup S_2) && \text{(D.3.8)} \end{aligned}$$

Message Sending and Pulsing

$$\begin{aligned} &\llbracket \text{send}_J \text{ a} \rrbracket(S_1) \cup \llbracket \text{send}_J \text{ a} \rrbracket(S_2) \\ &= (S_1 \cdot \wr I \xrightarrow{a} J \mid) \cup (S_2 \cdot \wr I \xrightarrow{a} J \mid) && \text{(D.3.8)} \\ &= (S_1 \cup S_2) \cdot \wr I \xrightarrow{a} J \mid && \text{(C.4.3)} \\ &= \llbracket \text{send}_J \text{ a} \rrbracket(S_1 \cup S_2) && \text{(D.3.8)} \end{aligned}$$

The cases for $\text{send}_J \text{ b}$ and pulse are completely analogous.

We now have shown that for all non-composed statements Lemma L.4.1 holds, which concludes the proof for the induction basis.

Induction Hypothesis

As our induction hypothesis we assume that Lemma L.4.1 holds for the LOOP programs P_1 , P_2 and P_3 . More formally we assume that the following equalities hold: $\llbracket P_1 \rrbracket(S_1) \cup \llbracket P_1 \rrbracket(S_2) = \llbracket P_1 \rrbracket(S_1 \cup S_2)$, $\llbracket P_2 \rrbracket(S_1) \cup \llbracket P_2 \rrbracket(S_2) = \llbracket P_2 \rrbracket(S_1 \cup S_2)$ and $\llbracket P_3 \rrbracket(S_1) \cup \llbracket P_3 \rrbracket(S_2) = \llbracket P_3 \rrbracket(S_1 \cup S_2)$.

Induction Step

We now prove that, given the induction hypothesis, Lemma L.4.1 holds for every composed statement:

Message Reception

$$\begin{aligned}
 & \llbracket \text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END} \rrbracket (S_1) \\
 & \quad \cup \llbracket \text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END} \rrbracket (S_2) \\
 = & \llbracket P_1 \rrbracket (S_1 \cdot |J \xrightarrow{a} I \hat{\circ}) \cup \llbracket P_2 \rrbracket (S_1 \cdot |J \xrightarrow{b} I \hat{\circ}) \cup \llbracket P_3 \rrbracket (S_1 \cdot |J \xrightarrow{\perp} I \hat{\circ}) \quad (\text{D.3.8}) \\
 & \quad \cup \llbracket P_1 \rrbracket (S_2 \cdot |J \xrightarrow{a} I \hat{\circ}) \cup \llbracket P_2 \rrbracket (S_2 \cdot |J \xrightarrow{b} I \hat{\circ}) \cup \llbracket P_3 \rrbracket (S_2 \cdot |J \xrightarrow{\perp} I \hat{\circ}) \\
 = & \llbracket P_1 \rrbracket (S_1 \cdot |J \xrightarrow{a} I \hat{\circ} \cup S_2 \cdot |J \xrightarrow{a} I \hat{\circ}) \quad (\text{I.H.}) \\
 & \quad \cup \llbracket P_2 \rrbracket (S_1 \cdot |J \xrightarrow{b} I \hat{\circ} \cup S_2 \cdot |J \xrightarrow{b} I \hat{\circ}) \\
 & \quad \cup \llbracket P_3 \rrbracket (S_1 \cdot |J \xrightarrow{\perp} I \hat{\circ} \cup S_2 \cdot |J \xrightarrow{\perp} I \hat{\circ}) \\
 = & \llbracket P_1 \rrbracket ((S_1 \cup S_2) \cdot |J \xrightarrow{a} I \hat{\circ}) \quad (\text{C.4.3}) \\
 & \quad \cup \llbracket P_2 \rrbracket ((S_1 \cup S_2) \cdot |J \xrightarrow{b} I \hat{\circ}) \\
 & \quad \cup \llbracket P_3 \rrbracket ((S_1 \cup S_2) \cdot |J \xrightarrow{\perp} I \hat{\circ}) \\
 = & \llbracket \text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END} \rrbracket (S_1 \cup S_2) \quad (\text{D.3.8})
 \end{aligned}$$

Concatenation

$$\begin{aligned}
 & \llbracket P_1 ; P_2 \rrbracket (S_1) \cup \llbracket P_1 ; P_2 \rrbracket (S_2) \\
 = & \llbracket P_2 \rrbracket (\llbracket P_1 \rrbracket (S_1)) \cup \llbracket P_2 \rrbracket (\llbracket P_1 \rrbracket (S_2)) \quad (\text{D.3.8}) \\
 = & \llbracket P_2 \rrbracket (\llbracket P_1 \rrbracket (S_1) \cup \llbracket P_1 \rrbracket (S_2)) \quad (\text{I.H.}) \\
 = & \llbracket P_2 \rrbracket (\llbracket P_1 \rrbracket (S_1 \cup S_2)) \quad (\text{I.H.}) \\
 = & \llbracket P_1 ; P_2 \rrbracket (S_1 \cup S_2) \quad (\text{D.3.8})
 \end{aligned}$$

Loops

$$\begin{aligned}
 & \llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket (S_1) \cup \llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket (S_2) \\
 = & \bigcup_{(\sigma, w) \in S_1} \left(\llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, w)\}) \right) \cup \bigcup_{(\sigma, w) \in S_2} \left(\llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, w)\}) \right) \quad (\text{D.3.8}) \\
 = & \bigcup_{(\sigma, w) \in S_1 \cup S_2} \left(\llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, w)\}) \right) \\
 = & \llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket (S_1 \cup S_2) \quad (\text{D.3.8})
 \end{aligned}$$

ω -Loops

$$\begin{aligned}
& \llbracket \text{LOOP } \omega \text{ DO } P_1 \text{ END} \rrbracket(S_1) \cup \llbracket \text{LOOP } \omega \text{ DO } P_1 \text{ END} \rrbracket(S_2) \\
&= \left\{ (\perp, w) \left| \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in \llbracket P_1 \rrbracket^k(S_1) \end{array} \right. \right\} \quad (\text{D.3.8}) \\
&\quad \cup \left\{ (\perp, w) \left| \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in \llbracket P_1 \rrbracket^k(S_2) \end{array} \right. \right\} \\
&= \left\{ (\perp, w) \left| \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in \llbracket P_1 \rrbracket^k(S_1) \cup \llbracket P_1 \rrbracket^k(S_2) \end{array} \right. \right\}
\end{aligned}$$

The expression $\llbracket P_1 \rrbracket^k(S_i)$ represents a k -fold application of P_1 to S_i which is the same as $\underbrace{\llbracket P_1; \dots; P_1 \rrbracket}_{k \text{ times}}(S_i)$ for $i \in \{1, 2\}$, so we can use the case for the concatenation:

$$\begin{aligned}
&= \left\{ (\perp, w) \left| \begin{array}{l} w \in \Sigma_{I/O}^\omega, \exists w_0 \cdot w_1 \cdot w_2 \cdots = w, \\ \text{such that } w_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w_0 \cdot w_1 \cdots w_k) \in \llbracket P_1 \rrbracket^k(S_1 \cup S_2) \end{array} \right. \right\} \\
&= \llbracket \text{LOOP } \omega \text{ DO } P_1 \text{ END} \rrbracket(S_1 \cup S_2) \quad (\text{D.3.8})
\end{aligned}$$

This concludes the proof for Lemma L.4.1 \square

From Lemma L.4.1, we derive a simple yet very useful corollary. It links the union of LOOP or LOOP+ ω configurations to the union of the corresponding LOOP- or LOOP+ ω -generated languages.

Corollary C.4.4:

For any LOOP or LOOP+ ω program $P \in \mathbb{L} \cup \mathbb{L}_\omega$ and any two configurations $S_1, S_2 \subseteq \mathbb{S}$, the following equality holds:

$$L_{S_1}(P) \cup L_{S_2}(P) = L_{S_1 \cup S_2}(P)$$

Proof:

$$\begin{aligned}
& L_{S_1}(P) \cup L_{S_2}(P) \\
&= \{w \mid (\sigma, w) \in \llbracket P \rrbracket(S_1)\} \cup \{w \mid (\sigma, w) \in \llbracket P \rrbracket(S_2)\} \quad (\text{D.3.10}) \\
&= \{w \mid (\sigma, w) \in (\llbracket P \rrbracket(S_1) \cup \llbracket P \rrbracket(S_2))\} \\
&= \{w \mid (\sigma, w) \in (\llbracket P \rrbracket(S_1 \cup S_2))\} \quad (\text{L.4.1}) \\
&= L_{S_1 \cup S_2}(P) \quad (\text{D.3.10})
\end{aligned}$$

\square

4.3 Prefix Preservation by the Application of LOOP or LOOP+ ω Semantics

We establish an important Lemma which states that the semantics of a LOOP or LOOP+ ω program preserves a common prefix of a set of configurations. This lemma will also be needed for later proving the main Theorem in Chapter 6.

Lemma L.4.2 (PREFIX PRESERVATION BY LOOP AND LOOP+ ω SEMANTICS):

For any LOOP or LOOP+ ω program $P \in \mathbb{L} \cup \mathbb{L}_\omega$, any set of configurations created by LOOP programs $S \subseteq \mathbb{S}$ and any word $w \in \Sigma_{I/O}^$, it holds that*

$$\llbracket P \rrbracket(w \cdot S) = w \cdot \llbracket P \rrbracket(S).$$

Proof: The proof of Lemma L.4.2 is again done by structural induction over all LOOP and LOOP+ ω programs, similarly to the proof of Lemma L.4.1. For the full proof of Lemma L.4.2, please refer to Appendix A.1. \square

Lemma L.4.2 is closely related to factoring out common prefixes from LOOP– and from LOOP+ ω –generated languages. Given Lemma L.4.2 we can easily conclude the following:

Corollary C.4.5:

For any LOOP or LOOP+ ω program $P \in \mathbb{L} \cup \mathbb{L}_\omega$, any set of configurations generated by a LOOP program $S \subseteq \mathbb{S}$ and any finite word $w \in \Sigma_{I/O}^$ it holds that*

$$L_{w.S}(P) = w \cdot L_S(P)$$

Proof:

$$\begin{aligned} L_{w.S}(P) &= \{v \mid (\sigma, v) \in \llbracket P \rrbracket(w \cdot S)\} && \text{(D.3.10)} \\ &= \{v \mid (\sigma, v) \in w \cdot \llbracket P \rrbracket(S)\} && \text{(L.4.2)} \\ &= \{v \mid (\sigma, v) \in \{(\tau, w \cdot z) \mid (\tau, z) \in \llbracket P \rrbracket(S)\}\} && \text{(D.3.5)} \\ &= \{w \cdot z \mid (\sigma, w \cdot z) \in \{(\tau, w \cdot z) \mid (\tau, z) \in \llbracket P \rrbracket(S)\}\} && (v \hat{=} w \cdot z) \\ &= w \cdot \{z \mid (\sigma, z) \in \{(\tau, z) \mid (\tau, z) \in \llbracket P \rrbracket(S)\}\} \\ &= w \cdot \{z \mid (\sigma, z) \in \llbracket P \rrbracket(S)\} \\ &= w \cdot L_S(P) && \text{(D.3.10)} \end{aligned}$$

\square

4.4 Miscellaneous Properties of LOOP– and LOOP+ ω –Generated Languages

In order to shorten upcoming proofs in the following chapters, a lemma on the languages of LOOP and LOOP+ ω programs is introduced which simplifies the expression $L_S(P)$ for some programs P :

Lemma L.4.3:

Let P_1, P_2 and P_3 be LOOP programs and let S be any configuration. Then the following holds:

- (I) $L_S(\text{skip}) = L(S)$
- (II) $L_S(x_i := x_j + 1) = L(S)$
- (III) $L_S(\text{send}_J \ a) = L(S) \cdot \wr I \xrightarrow{a} J |$ (analogous for $\text{send}_J \ b$ and pulse)
- (IV) $L_S(\text{RECV}_J \ a \ \text{DO} \ P_1 \ \text{OR} \ b \ \text{DO} \ P_2 \ \text{OR} \ _ \ \text{DO} \ P_3 \ \text{END})$
 $= L_{S \cdot |J \xrightarrow{a} I}^\circ(P_1) \cup L_{S \cdot |J \xrightarrow{b} I}^\circ(P_2) \cup L_{S \cdot |J \xrightarrow{_} I}^\circ(P_3)$
- (V) $L_S(P_1; P_2) = \bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S)} \left(v \cdot L_{\{(\tau, \varepsilon)\}}(P_2) \right)$
- (VI) $L_S(\text{LOOP} \ \omega \ \text{DO} \ P_1 \ \text{END}) = \lim \left(\bigcup_{k \in \mathbb{N}} L_S(\underbrace{P_1; \dots; P_1}_{k \text{ times}}) \right)$

Proof of (I):

$$L_S(\text{skip}) = \{w \mid (\sigma, w) \in \llbracket \text{skip} \rrbracket(S)\} \quad (\text{D.3.10})$$

$$= \{w \mid (\sigma, w) \in S\} \quad (\text{D.3.8})$$

$$= L(S) \quad (\text{D.3.9})$$

Proof of (II)

$$\begin{aligned} & L_S(x_i := x_j + c) \\ &= \{w \mid (\sigma, w) \in \llbracket x_i := x_j + c \rrbracket(S)\} \end{aligned} \quad (\text{D.3.10})$$

$$= \{w \mid (\sigma, w) \in \{(\tau[x_i \leftarrow \tau(x_j) + c], w) \mid (\tau, w) \in S\}\} \quad (\text{D.3.8})$$

$$= \{w \mid (\tau, w) \in S\}$$

$$= L(S) \quad (\text{D.3.9})$$

Proof of (III)

$$L_S(\text{send}_J \ a) = \{w \mid (\sigma, w) \in \llbracket \text{send}_J \ a \rrbracket(S)\} \quad (\text{D.3.10})$$

$$= \left\{ w \mid (\sigma, w) \in S \cdot \wr I \xrightarrow{a} J | \right\} \quad (\text{D.3.8})$$

$$= \left\{ w \mid (\sigma, w) \in \left\{ \left(\tau, v \cdot \wr I \xrightarrow{a} J | \right) \mid (\tau, v) \in S \right\} \right\} \quad (\text{D.3.5})$$

$$= \left\{ v \cdot \wr I \xrightarrow{a} J | \mid (\tau, v) \in S \right\}$$

$$= \{v \mid (\tau, v) \in S\} \cdot \wr I \xrightarrow{a} J | \quad (\text{D.3.5})$$

$$= L(S) \cdot \wr I \xrightarrow{a} J | \quad (\text{D.3.9})$$

The case for $\text{send}_J \ b$ and pulse is completely analogous.

Proof of (IV)

$$\begin{aligned}
 & L_S(\text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END}) \\
 &= \{w \mid (\sigma, w) \in \llbracket \text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END} \rrbracket(S)\} \quad (\text{D.3.10}) \\
 &= \left\{ w \mid (\sigma, w) \in \llbracket P_1 \rrbracket(S \cdot |J \xrightarrow{a} I^\circ) \cup \llbracket P_2 \rrbracket(S \cdot |J \xrightarrow{b} I^\circ) \right. \\
 &\quad \left. \cup \llbracket P_3 \rrbracket(S \cdot |J \xrightarrow{\perp} I^\circ) \right\} \quad (\text{D.3.8}) \\
 &= \left\{ w \mid (\sigma, w) \in \llbracket P_1 \rrbracket(S \cdot |J \xrightarrow{a} I^\circ) \right\} \cup \left\{ w \mid (\sigma, w) \in \llbracket P_2 \rrbracket(S \cdot |J \xrightarrow{b} I^\circ) \right\} \\
 &\quad \cup \left\{ w \mid (\sigma, w) \in \llbracket P_3 \rrbracket(S \cdot |J \xrightarrow{\perp} I^\circ) \right\} \\
 &= L_{S \cdot |J \xrightarrow{a} I^\circ}(P_1) \cup L_{S \cdot |J \xrightarrow{b} I^\circ}(P_2) \cup L_{S \cdot |J \xrightarrow{\perp} I^\circ}(P_3) \quad (\text{D.3.10})
 \end{aligned}$$

Proof of (V)

$$\begin{aligned}
 & L_S(P_1; P_2) \\
 &= \{w \mid (\sigma, w) \in \llbracket P_1; P_2 \rrbracket(S)\} \quad (\text{D.3.10}) \\
 &= \{w \mid (\sigma, w) \in \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(S))\} \quad (\text{D.3.8}) \\
 &= \left\{ w \mid (\sigma, w) \in \llbracket P_2 \rrbracket \left(\bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S)} \left(\{(\tau, v)\} \right) \right) \right\} \\
 &= \left\{ w \mid (\sigma, w) \in \left(\bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S)} \left(\llbracket P_2 \rrbracket(\{(\tau, v)\}) \right) \right) \right\} \quad (\text{L.4.1}) \\
 &= \left\{ w \mid (\sigma, w) \in \left(\bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S)} \left(v \cdot \llbracket P_2 \rrbracket(\{(\tau, \varepsilon)\}) \right) \right) \right\} \quad (\text{L.4.2}) \\
 &= \bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S)} \left(v \cdot L_{\{(\tau, \varepsilon)\}}(P_2) \right)
 \end{aligned}$$

Proof of (VI)

$$\begin{aligned}
 & L_S(\text{LOOP } \omega \text{ DO } P_1 \text{ END}) \\
 &= \{w \mid (\sigma, w) \in \llbracket \text{LOOP } \omega \text{ DO } P_1 \text{ END} \rrbracket(S)\} \quad (\text{D.3.10}) \\
 &= \left\{ w \mid (\sigma, w) \in \left\{ (\perp, v) \mid \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \llbracket P_1 \rrbracket^k(S) \end{array} \right\} \right\} \quad (\text{D.3.8}) \\
 &= \left\{ v \mid \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \llbracket P_1 \rrbracket^k(S) \end{array} \right\}
 \end{aligned}$$

$$\begin{aligned}
&= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \underbrace{\llbracket P_1; \dots; P_1 \rrbracket(S)}_{k \text{ times}} \end{array} \right. \right\} \\
&= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \\ \text{it holds that } v_0 \cdot v_1 \cdots v_k \in L_S(\underbrace{P_1; \dots; P_1}_{k \text{ times}}) \end{array} \right. \right\} \quad (\text{D.3.10}) \\
&= \left\{ v \left| \begin{array}{l} \text{there exist infinitely many prefixes } v' \text{ of } v, \\ \text{such that } v' \in \bigcup_{k \in \mathbb{N}} L_S(\underbrace{P_1; \dots; P_1}_{k \text{ times}}) \end{array} \right. \right\} \\
&= \left\{ v \left| \left| \text{Pref}(v) \cap \left(\bigcup_{k \in \mathbb{N}} L_S(\underbrace{P_1; \dots; P_1}_{k \text{ times}}) \right) \right| = \infty \right. \right\} \\
&= \lim \left(\bigcup_{k \in \mathbb{N}} L_S(\underbrace{P_1; \dots; P_1}_{k \text{ times}}) \right) \quad \square
\end{aligned}$$

Furthermore we introduce a corollary which provides a special case in which the concatenation of words and configurations as in Definition D.3.5 is commutative. This special case is the case that $L(S) = \{\varepsilon\}$:

Corollary C.4.6:

Let $S_\varepsilon \subseteq \mathbb{S}_\perp$ such that $L(S_\varepsilon) = \{\varepsilon\}$, then the following equality holds for any word w :

$$S_\varepsilon \cdot w = w \cdot S_\varepsilon$$

Proof: First we observe that any such set of configurations S_ε with $L(S_\varepsilon) = \{\varepsilon\}$ is obviously of the form $S_\varepsilon = \{(\sigma, \varepsilon) \mid (\sigma, \varepsilon) \in S_\varepsilon\}$. Then we conclude

$$\begin{aligned}
S_\varepsilon \cdot w &= \{(\sigma, \varepsilon) \mid (\sigma, \varepsilon) \in S_\varepsilon\} \cdot w \\
&= \{(\sigma, \varepsilon \cdot w) \mid (\sigma, \varepsilon) \in S_\varepsilon\} \quad (\text{D.3.5})
\end{aligned}$$

$$\begin{aligned}
&= \{(\sigma, w) \mid (\sigma, \varepsilon) \in S_\varepsilon\} \\
&= \{(\sigma, w \cdot \varepsilon) \mid (\sigma, \varepsilon) \in S_\varepsilon\} \\
&= w \cdot \{(\sigma, \varepsilon) \mid (\sigma, \varepsilon) \in S_\varepsilon\} \quad (\text{D.3.5}) \\
&= w \cdot S_\varepsilon. \quad \square
\end{aligned}$$

4.5 Reactiveness of LOOP– and LOOP+ ω –Generated Languages

Another interesting property of the languages generated by LOOP and LOOP+ ω programs is that they behave in some sense non-deterministic with respect to the input symbols $|J \xrightarrow{a} I\rangle$, $|J \xrightarrow{b} I\rangle$ and $|J \xrightarrow{\perp} I\rangle$, and on the other hand strictly

deterministic with respect to the output symbols $\langle I \xrightarrow{a} J \rangle$, $\langle I \xrightarrow{b} J \rangle$ and \sim^I . This is due to the fact that if the program decides to read a symbol from the input channel, then it cannot make any assumption which symbol it reads or whether the channel is empty. Every one of these cases must be processed. On the other hand, the program's state depends solely on the initial state and the history of the inputs. So given a certain initial state and a certain history of inputs, the output generated next by the program is completely determined.

In the following we will formalize this property which we call *reactiveness*. Thereafter we will prove that every LOOP- and every LOOP+ ω -generated language is reactive.

Definition D.4.3 (REACTIVENESS OF A LANGUAGE):

A language L over an alphabet Σ is called **reactive**, if there exists a partition $\Sigma_{\text{in}} \dot{\cup} \Sigma_{\text{out}} = \Sigma$, such that the following two properties hold:

1. **Input-Enabledness:** If for some input symbol $a \in \Sigma_{\text{in}}$ it holds that $\alpha \cdot a \cdot \beta \in L$, then for all input symbols $b \in \Sigma_{\text{in}}$ there exists a word ζ such that $\alpha \cdot b \cdot \zeta \in L$.
2. **Output-Determinedness:** If for some output symbol $a \in \Sigma_{\text{out}}$ it holds that $\alpha \cdot a \cdot \beta \in L$, then all words with prefix α in L have the prefix $\alpha \cdot a$.

Intuitively this means that (1.) if the next symbol in a word of a language is an input symbol a , then every other input symbol b could have appeared at this position as well, and (2.) given a history of symbols α if the next symbol is an output symbol a , then no other output symbol b could have been produced as the output is determined completely by the history α . We now prove that every LOOP- and every LOOP+ ω -generated language satisfies the property of being reactive:

Lemma L.4.4 (REACTIVENESS OF LOOP- AND LOOP+ ω -GENERATED LANGUAGES):

All LOOP- and LOOP+ ω -generated languages are reactive, i.e. for all programs P the language $L(P)$ is reactive.

We prove Lemma L.4.4 by proving the following, more general variant of Lemma L.4.4:

Lemma L.4.5:

Let P be a LOOP or LOOP+ ω program and let $S_\varepsilon = \{(\sigma, \varepsilon)\}$ be a configuration containing only one valuation-word-pair where the word is the empty word. Then $L_{S_\varepsilon}(P)$ is reactive.

If Lemma L.4.5 is true, then obviously Lemma L.4.4 holds as well.

Proof of Lemma L.4.5: The proof of Lemma L.4.5 is again done by structural induction over all LOOP and all LOOP+ ω programs, similarly to the proof of Lemma L.4.1. For the full proof of Lemma L.4.5, please refer to Appendix A.2. \square

In the following, we use the the property of reactiveness to show that every LOOP- and every LOOP+ ω -generated language has a common maximal prefix of pulse symbols. For that, we first define a function which strips away all leading pulse symbols from every word in a language:

Definition D.4.4 (THE strip-FUNCTION):

Let $\text{strip}(w)$ be a function which removes all leading pulse symbols from the word w , so

$$\text{strip}(w) = \begin{cases} \text{strip}(v), & \text{if } w = \overset{I}{\sim} \cdot v \\ w, & \text{else} \end{cases}$$

Furthermore let $\text{strip}(L) = \{\text{strip}(w) \mid w \in L\}$ for a language L and let $\text{strip}(\mathcal{L}) = \{\text{strip}(L) \mid L \in \mathcal{L}\}$ for a set of languages \mathcal{L} .

Using the strip-function and Lemma L.4.4, we can now prove the desired property:

Corollary C.4.7:

For each LOOP- or LOOP+ ω -generated language $L(P)$ there exists $n \in \mathbb{N}$ such that $L(P) = \{\overset{I}{\sim}\}^n \cdot \text{strip}(L(P))$.

Proof: Assume n is the length of the maximal common prefix of pulse symbols of every word in $L(P)$. We can then rewrite $L(P)$ as $L(P) = \{\overset{I}{\sim}\}^n \cdot L$, for some suited language L . It remains to show that $L = \text{strip}(L(P))$. For that assume the opposite, i.e. assume there existed a word $\overset{I}{\sim} \cdot w \in L$. Due to the reactiveness of L (cf. Lemma L.4.4 and Definition D.4.3) we then know that every other word in L also starts with a pulse symbol. This however is a contradiction to the fact that we have factored out the maximal common prefix of pulse symbols. Thus the assumption is wrong and L does indeed not contain any word with a leading pulse symbol, thus $L = \text{strip}(L(P))$, and thus $L(P) = \{\overset{I}{\sim}\}^n \cdot \text{strip}(L(P))$. \square

5

Decision Problems for $\mathcal{L}oop_{\omega}$

In this chapter, we study the expressive power of $\text{LOOP}+\omega$ programs and the decidability of various decision problems related to the analysis of the communication behavior of $\text{LOOP}+\omega$ programs. The idea behind a language like $\text{LOOP}+\omega$ was to reduce expressiveness of programs by *guaranteeing* non-termination and using only primitive recursive — thus terminating — loop bodies. Our hope was that some properties regarding the communication behavior become decidable through that, unlike in the case of ordinary **WHILE** programs where programs may or may not terminate.

As we will learn in this chapter, our hopes are dashed: We show that $\text{LOOP}+\omega$ programs may in some sense simulate any **WHILE** program which is why many decision problems for $\text{LOOP}+\omega$ programs unfortunately remain undecidable. We also show in which way $\mathcal{L}oop_{\omega}$ — the class of $\text{LOOP}+\omega$ -generated languages — agrees in some sense with the class of recursively enumerable languages. Furthermore, we present an adaption of Rice's Theorem to $\text{LOOP}+\omega$ programs which is not only an interesting result in itself, but in addition also useful for proving the undecidability of some decision problems for $\text{LOOP}+\omega$ programs.

A greater part of this chapter deals with the decidability of some decision problems for $\mathcal{L}oop_{\omega}$, which we found interesting in the context of verification of such programs. Namely, those are:

- **The Emptiness Problem:** Is the language generated by a $\text{LOOP}+\omega$ program empty?
- **The Intersection Problem:** Is the intersection of a language generated by a $\text{LOOP}+\omega$ program with some ω -regular language empty?
- **The Equivalence Problem:** Are the languages of two $\text{LOOP}+\omega$ programs equal?
- **The Inclusion Problem:** Is the language of one $\text{LOOP}+\omega$ program a subset of the language of a second $\text{LOOP}+\omega$ program?

- **The Regularity Problem:** Is the language of a $\text{LOOP}+\omega$ program ω -regular?
- **The Pulse–Distance Boundedness Problem:** Is the distance between any two pulse symbols bounded for every word of the language of a $\text{LOOP}+\omega$ program?
- **The ω –Power Problem:** Is the language of a $\text{LOOP}+\omega$ program of the form $L_1 \cdot L_2^\omega$ for some two languages L_1 and L_2 ?

Recall that we say that a problem P is decidable for the class of $\text{LOOP}+\omega$ -generated languages Loop_ω , if there exists an algorithm \mathfrak{A}_P such that for any language $L \in \text{Loop}_\omega$ effectively given by some $\text{LOOP}+\omega$ program P (i.e. $L(P) = L$) the question whether L satisfies the problem statement of P can be answered by \mathfrak{A}_P correctly for every input program in finite time. We will show that basically all of the above decision problems are undecidable for $\text{LOOP}+\omega$ programs in the following sections, so we have:

Theorem T.I (DECIDABILITY OF PROBLEMS FOR Loop_ω):

The Equivalence Problem, the Inclusion Problem, the Regularity Problem, the Pulse–Distance Boundedness Problem and the ω –Power Problem are undecidable for the class of $\text{LOOP}+\omega$ -generated languages.

The Emptiness Problem for the class of $\text{LOOP}+\omega$ -generated languages is trivially decidable, however, a more interesting variant of it — the Essential Emptiness Problem — is undecidable.

5.1 Recursive Functions

Before we study the undecidability of the above problems, we give a very short recall on recursive functions. We will need some insight into recursive functions in order to obtain results on the expressiveness of $\text{LOOP}+\omega$ programs.

There are two classes of functions computable by machines, algorithms, etc. which are especially well studied: the recursive functions and the primitive recursive functions. For a better distinction, we shall refer to the former as to the *general* recursive functions. All primitive recursive functions are also general recursive but the converse does not hold in general.

It is known that the class of primitive recursive functions resembles exactly the functions which can be computed by a LOOP program [MR67]. Nearly all functions which appear in practice are actually primitive recursive. Amongst those one can find simple functions like addition, multiplication, division, potentiation, etc. But also much more complicated functions like calculating the x^{th} prime number or the first x digits of $\sin(1/2)$ are both primitive recursive. As Albert R. Meyer and Dennis M. Ritchie state in [MR67], "a fairly careful analysis of the definition of LOOP programs is required in order to discover a function which they cannot compute". Such an example for a general recursive, yet not primitive recursive function is the extremely fast growing Ackermann–Function named after Wilhelm Ackermann who first published it in 1928 [Ack28].

For the general recursive functions, i.e. according to the Church–Turing Thesis the so to say “intuitively” computable functions, there exist numerous equivalent characterizations, including μ -recursive functions, λ -Calculus or Turing Machines. Conveniently, there is also an equivalent characterization by means of a minimalistic model programming language, the WHILE programming language. That means that every general recursive function can be computed by some WHILE program [Sch00]. In the following we discuss syntax and semantics of WHILE programs as well as some fundamental results due to Alan M. Turing and Stephen C. Kleene.

5.1.1 WHILE Programs

In this section, we recall the syntax and very briefly the semantics of WHILE programs. We start off with the definition of their syntax:

Definition D.5.1 (SYNTAX OF WHILE PROGRAMS):

The syntax of WHILE programs is defined by the following grammar:

$$\begin{aligned}
 P &\longrightarrow \text{skip} \\
 &\quad | x_i := x_j + c \\
 &\quad | P; P \\
 &\quad | \text{LOOP } x \text{ DO } P \text{ END} \\
 &\quad | \text{WHILE } x \neq 0 \text{ DO } P \text{ END}
 \end{aligned}$$

We denote the set of all WHILE programs which are generated by the grammar above by \mathbb{W} .

Note that every LOOP program is also a WHILE program, i.e. $\mathbb{L} \subseteq \mathbb{W}$. When comparing Definition D.5.1 to Definition D.3.1 we observe that merely the while-loop was added.

As we are by now familiar with the semantics of LOOP programs and we will not be needing the details of the semantics of WHILE in the remainder of this thesis, we will discuss the semantics of WHILE only briefly: The semantics of WHILE programs is basically the same as the semantics of the LOOP programs. Only the while-loop needs additional explanation: In a while-loop `WHILE $x \neq 0$ DO P END` the loop body P is executed as many times, as the program variable x is unequal to 0 before the next execution of P . Formally, we could define the semantics of a while-loop as follows:

$$\langle \text{WHILE } x \neq 0 \text{ DO } P \text{ END} \rangle(\sigma) = \begin{cases} \sigma, & \text{if } \sigma(x) = 0 \\ \langle P; \text{WHILE } x \neq 0 \text{ DO } P \text{ END} \rangle(\sigma), & \text{else} \end{cases}$$

Potentially, the program variable x might never be set to 0 no matter how many times P is executed and therefore, while-loops do not necessarily terminate. In these cases of non-termination the evaluation of $\langle \text{WHILE } x \neq 0 \text{ DO } P \text{ END} \rangle(\sigma)$ also does not terminate and $\langle \text{WHILE } x \neq 0 \text{ DO } P \text{ END} \rangle(\sigma) = \langle P \rangle(\langle P \rangle(\langle P \rangle(\langle P \rangle(\langle P \rangle(\dots))))))$ is actually undefined. Therefore we define in these cases that $\langle \text{WHILE } x \neq 0 \text{ DO } P \text{ END} \rangle(\sigma)$ evaluates to \perp , where \perp represents non-termination. We also define that $\langle P \rangle(\perp) = \perp$ for any WHILE program P . Thus WHILE programs define *partial* functions.

5.1.2 The Halting Problem

It is a very interesting and relevant question in software verification whether it holds for a WHILE program P that $\langle P \rangle(\sigma) \neq \perp$, for a fixed input σ_0 . From an operational point of view this is equivalent to the question whether the execution of a WHILE program P ever terminates on the input σ_0 . This problem is known as the Halting Problem:

Problem P.5.1 (THE HALTING PROBLEM FOR WHILE PROGRAMS):

Given: A WHILE program $P \in \mathbb{W}$.

Question: Does P halt on input σ_0 , i.e. does it hold that $\langle P \rangle(\sigma_0) \neq \perp$?

We will refer to halting and terminating synonymously. As Turing showed in 1936, the Halting Problem for Turing Machines is undecidable [Tur36]. That means that no algorithm exists which can decide for any given Turing Machine M whether M halts on a certain input. From that it follows that the Halting Problem for WHILE programs (Problem P.5.1) is also undecidable, as Turing Machines and WHILE programs are equivalent models of computation, so we get:

Remark R.5.1 (UNDECIDABILITY OF THE HALTING PROBLEM [Tur36]):

The Halting Problem for WHILE programs is undecidable.

That means that there exists no algorithm which can decide for any given WHILE program P whether P halts on input σ_0 . We will later use this fact later for proving the undecidability of other problems by reducing the Halting Problem to the problem in question. If we can successfully reduce the Halting Problem to a problem Q , the problem Q cannot be decidable because otherwise we could effectively decide the Halting Problem by using the decision algorithm for Q which, however, is a contradiction to the undecidability of the Halting Problem.

5.1.3 The Kleene Normal Form Theorem

WHILE programs and therefore all general recursive functions have a finite representation and can therefore be numbered. Such a numbering of the WHILE programs or general recursive functions is commonly known as a *gödelization* or *Gödel Numbering*.

Remark R.5.2 (GÖDEL NUMBERING):

There exists a bijective mapping $\gamma: \mathbb{W} \rightarrow \mathbb{N}$ and thereby there exists a bijective mapping from the general recursive functions to the natural numbers, too.

$\gamma(P)$ ($\gamma(f)$, respectively) is called the **Gödel Number** of a WHILE program P (of a general recursive function f , respectively).

An important result due to Kleene is that every general recursive function $f(x)$ can be expressed by means of a universal μ -recursive function $u(\gamma(f), x)$, where u is constructed with only one single μ -operator [Kle38]. This result is known as the *Kleene Normal Form Theorem*.

As the μ -recursive functions and WHILE programs are equivalent models of computation and the μ -operator is the analogous concept to the while-loop, the Kleene Normal Form Theorem translates to WHILE programs as follows:

Remark R.5.3 (KLEENE NORMAL FORM THEOREM FOR WHILE PROGRAMS):

There exists a WHILE program U with only a single while-loop such that

$$U = x_k := 1; \text{ WHILE } x_k \neq 0 \text{ DO } T \text{ END},$$

T is a LOOP program and for every WHILE program P it holds that

$$\langle U \rangle(\sigma_{(\gamma(P), \vec{x})}) = \langle P \rangle(\sigma_{(\vec{x})}).$$

It is obvious that U or rather T reads the Gödel Number $\gamma(P)$ from the first program variable x_0 . For each WHILE program P we can now of course simply hard-code $\gamma(P)$ into T and thus we obtain for every WHILE program P a completely equivalent WHILE program in Kleene Normal Form, i.e. with only a single while-loop:

Remark R.5.4 (KLEENE NORMAL FORM OF WHILE PROGRAMS):

Every WHILE program P can be transformed into an equivalent program of the form

$$x_k := 1; \text{ WHILE } x_k \neq 0 \text{ DO } P' \text{ END},$$

where x_k does not occur in P and P' is a LOOP program. We say that a program is in Kleene Normal Form, if it is in the above form.

As the while-loop body of WHILE programs in Kleene Normal Form is a LOOP program whose execution of course always terminates, but WHILE programs in general do not necessarily halt, it is obviously that the following holds:

Corollary C.5.8 (TERMINATION OF WHILE PROGRAMS IN KLEENE NORMAL FORM):

Let P be a WHILE program in Kleene Normal Form, i.e.

$$P = x_k := 1; \text{ WHILE } x_k \neq 0 \text{ DO } P' \text{ END}$$

Then P terminates, if and only if after finitely many executions of the loop body P' the program variable x_k is set to 0.

This leads to the conclusion that an algorithm for deciding whether the variable x_k is ever set to 0 when executing a WHILE program in Kleene Normal Form can be used to decide the (undecidable) Halting Problem for WHILE programs.

5.2 Expressiveness of LOOP+ ω

In this section, we will show two things: First, we will show that LOOP+ ω has enough expressive power to simulate the execution of a WHILE program and generate a language depending solely on the halting behavior of the simulated WHILE program. Secondly, we discuss that LOOP+ ω is expressive enough to express every recursively enumerable language.

5.2.1 Simulation of WHILE-Programs in $\text{LOOP}+\omega$

In this section we discuss how to construct for every WHILE program P a $\text{LOOP}+\omega$ program O_P , which simulates the execution of P and whose language $L(O_P)$ depends solely on the halting behavior of P . Namely, O_P will produce a certain word if P halts and a different word if P does not halt. We call the program O_P the $\text{LOOP}+\omega$ simulation of P . For a WHILE program P in Kleene Normal Form (otherwise, transform P into Kleene Normal Form first) the $\text{LOOP}+\omega$ simulation of P can be constructed as follows:

Definition D.5.2 (LOOP+ ω SIMULATION OF WHILE PROGRAMS):

Let P be a WHILE program in Kleene Normal Form, i.e.

$$P = x_k := 1; \text{ WHILE } x_k \neq 0 \text{ DO } P' \text{ END.}$$

Then the $\text{LOOP}+\omega$ simulation O_P of P is the following $\text{LOOP}+\omega$ program:

```

1:   $x_k := 1;$ 
2:  LOOP  $\omega$  DO
3:      IF  $x_k \neq 0$  THEN
4:           $P'$ 
5:      ELSE
6:          sendJ a
7:      END;
8:      pulse
9:  END

```

As mentioned above, the language $L(O_P)$ of O_P depends solely on the halting behavior of P . This is stated by the following lemma:

Lemma L.5.1 (DEPENDENCY OF $L(O_P)$ ON THE HALTING BEHAVIOR OF P):

The following holds for the language $L(O_P)$ of the $\text{LOOP}+\omega$ simulation O_P of a WHILE program P :

$$L(O_P) = \begin{cases} \{\{\text{wavy line}\}^\omega\}, & \text{if } P \text{ does not halt} \\ \{\{\text{wavy line}\}^n \cdot \{I \xrightarrow{a} J \mid \cdot \text{wavy line}\}^\omega\}, & \text{if } P \text{ halts,} \end{cases}$$

where $n \in \mathbb{N}$ is the number of iterations, the loop body P' of the WHILE program P is executed before P halts, if P halts.

Proof: Assume P halts. According to C.5.8 that means that within finitely many executions of the loop body P' of the program P the variable x_k is set to 0 (recall that P is in Kleene Normal Form). If this is the case, then in the execution of O_P the variable x_k is also set to 0 after finitely many executions of the ω -loop body. Note that while $x_k \neq 0$ the ω -loop body simply executes P' followed by the obligatory pulse. So as long as the variable x_k evaluates to something other than 0, only the pulse is generated in the I/O language of O_P . As soon as x_k is set to zero, the variable x_k is never again touched and in all following executions of the

ω -loop body the symbol $\{I \xrightarrow{a} J\}$ followed by the obligatory pulse is generated ad infinitum. Hence the language of O_P is in this case given by

$$\underbrace{\{\underbrace{\sim\sim\sim^I}_{x_k \text{ not yet 0}}\}^n}_{x_k \text{ not yet 0}} \cdot \underbrace{\{\{I \xrightarrow{a} J\} \cdot \underbrace{\sim\sim\sim^I}_{x_k \text{ now 0}}\}^\omega}_{x_k \text{ now 0}}.$$

Now assume P does not halt. According to C.5.8 that means that P' will be executed infinitely many times without ever setting x_k to 0. If this is the case, then in the execution of O_P the variable x_k is also never set to 0. As long as the variable evaluates to something other than 0, only the pulse is generated in the I/O language. Since this is the case ad infinitum, as P does not halt and therefore x_k is never set to 0, the only symbol that is produced (infinitely many times) is the pulse symbol $\sim\sim\sim^I$. Hence the language of O_P is in this case given by

$$\{\sim\sim\sim^I\}^\omega. \quad \square$$

5.2.2 Enumeration of all Recursively Enumerable Sets in LOOP+ ω

The LOOP+ ω programming language is expressive enough to enumerate every recursively enumerable set of natural numbers. In fact, we can even construct a universal LOOP+ ω program E which in some sense is capable of enumerating *every* recursively enumerable set of natural numbers. As there exists a computable bijective mapping from the recursively enumerable sets of natural numbers to the recursively enumerable languages, we can thereby of course enumerate those as well. Here, however, we shall focus on recursively enumerable sets of natural numbers. In order to construct such a program, we need to know the following result from classical recursion theory:

Remark R.5.5 (PROJECTION THEOREM [Kle36]):

$M \subseteq N$ is recursively enumerable if and only if M is the domain of some general recursive function.¹

We will make use of the Projection Theorem to construct the universal LOOP+ ω program E which enumerates every recursively enumerable set of natural numbers. Before we present the construction of E , we recall the Kleene Normal Form Theorem: From that we know that there exists a universal WHILE program U which simulates the behavior of any WHILE program P on the input \vec{x} (cf. Remark R.5.3). If we interpret P as a *unary* function, which we shall do in the remainder of this section, this basically means that it simulates P on input x . Recall that U is of the form

$$U = x_k := 1; \text{ WHILE } x_k \neq 0 \text{ DO } T \text{ END}$$

and takes the Gödel Number $\gamma(P)$ and the number x as its input in order to simulate the WHILE program P on input x . Based on U , it is now very easy to construct a LOOP program U' which takes *three* numbers as its input: (1) a Gödel Number $\gamma(P)$ of a WHILE program P , (2) a number of steps k , and (3) some input x for P . U' shall operate as follows: It simulates the WHILE program P encoded by the

¹The result of Remark R.5.5 is sometimes also used as the definition of recursive enumerability.

Gödel Number $\gamma(P)$ on input x but — unlike U — it executes the subprogram T at most a finite number of k times. If P has halted within a maximum of k executions of T on input x , U' indicates that by setting a dedicated variable x_{halted} to 1. The construction of U' on the basis of U is easy: The main idea is basically to just replace the while-loop of U with a loop-loop which loops over k .

By means of U' we now have a LOOP program at hand which can help us check for a WHILE program P whether a number x is in the domain of the function calculated by P . In the following we shall denote the domain of the function calculated by a program P by $\text{dom}(P)$. It is very important to note that U' *cannot decide* whether $x \in \text{dom}(P)$. However, if in fact $x \in \text{dom}(P)$ holds, then there exists some number $k \in \mathbb{N}$ such that U' successfully indicates that $x \in \text{dom}(P)$.

With the Projection Theorem in mind and the LOOP program U' at hand, we can now construct the LOOP+ ω program E which enumerates every recursively enumerable set of natural numbers: E will read a Gödel Number from its input channel and is thereby told which recursively enumerable set it shall enumerate. Specifying such a set by means of a Gödel Number is possible due to the Projection Theorem: The theorem states that for every recursively enumerable set there exists an according general recursive function and thereby an according WHILE program. After the input of the Gödel Number, E checks for every natural number whether it is in the domain of the function which is calculated by the program encoded in the Gödel Number. This is done by invoking the LOOP program U' . Again, E does *not decide* whether any number is in that domain, but if a number x is in the domain it outputs x . As is it quite tedious to provide the whole construction of E as a LOOP+ ω program, we describe the steps of E only in principle:

1. Read a Gödel Number $\gamma(P)$ from the input channel.

The Gödel Number can be encoded by a binary encoding over the symbols a and b . As long as a 's and b 's are read from the channel, increase a counter dedicated to $\gamma(P)$ according to the binary encoding. As soon as the channel is empty, the input of the Gödel Number is finished.

2. Set x_k to 1.

3. Do infinitely many times (ω -loop):

- (a) Set x_x to 0.

- (b) Do x_k times:

- i. Set x_{halted} to 0.

- ii. Run U' on input $\gamma(P)$, x_k and x_x .

That means: simulate the program P on input x_x allowing U' to execute its subprogram T at most x_k times. If P has halted on input x_x within a maximum number of x_k executions of T , set x_{halted} to 1.

- iii. If x_{halted} is 1:

- A. Do x_x times: Send symbol a .

- B. Send symbol b .

- iv. Increase x_x by 1.

- (c) Increase x_k by 1.

We now describe how E operates in more detail: First a Gödel Number $\gamma(P)$ of a WHILE program P is read from the input channel where it is encoded by a binary encoding over the symbols a and b . Then x_k is set to 1.

Next, the ω -loop starts. In every iteration of the ω -loop, first the variable x_x is initialized with 0. Then the following is done for x_k times:

E sets x_{halted} to 0 and then runs the program U' to test whether the number x_x is in $\text{dom}(P)$. For that U' is allowed to execute its subprogram T a number of x_k times. If P halts on input x_x within x_k many executions of T , then U' indicates that x_x is in $\text{dom}(P)$ by setting x_{halted} to 1. Next, if U' has indicated that P has halted on x_x the following is done: E outputs the number x_x in a unary encoding over the symbols a , meaning that it sends out x_x many symbols a . After that, it terminates the output of x_x by sending the symbol b . Then x_x is increased by 1.

After x_k iterations, at the end of the ω -loop body x_k is increased by 1. The loop-body of the ω -loop is of course executed infinitely many times. So if some number x_x is in $\text{dom}(P)$ then eventually the number of allowed steps x_k will be big enough for U' to indicate this. Therefore every number in $\text{dom}(P)$ will eventually be output by E .

Note that the numbers output by E will be repeated and generally not be output in canonical order. This drawback, however, is inherent to recursively enumerable, yet non-recursive, sets. The language $L(E)$ generated by E is the union of all words of the form

$$\text{bin}(\gamma(P)) \cdot |J \xrightarrow{\perp} I \int \cdot \underbrace{\int I \xrightarrow{a} J | \dots \int I \xrightarrow{a} J |}_{x_1 \text{ times}} \cdot \int I \xrightarrow{b} J | \cdot \underbrace{\int I \xrightarrow{a} J | \dots \int I \xrightarrow{a} J |}_{x_2 \text{ times}} \cdot \int I \xrightarrow{b} J | \dots$$

where $\text{bin}(\gamma(P))$ is the binary encoding of a Gödel Number $\gamma(P)$ of a WHILE program P over the alphabet $\{|J \xrightarrow{a} I \int, |J \xrightarrow{b} I \int\}$ and each x_i is in $\text{dom}(P)$. In addition $L(E)$ also contains all words of the form

$$\{|J \xrightarrow{a} I \int, |J \xrightarrow{b} I \int\}^\omega.$$

The latter type of words in $L(E)$ represent infinite Gödel Numbers which are of course not meaningful. The former type of words, however, causes that $L(E)$ contains for every recursively enumerable set an ω -word which enumerates all numbers in that set, i.e. given a recursively enumerable set M , there exists an ω -word $w_M \in L(E)$, such that

- $\text{bin}(\gamma(P)) \cdot |J \xrightarrow{\perp} I \int$ with $M = \text{dom}(P)$ is a prefix of w_M , and
- for every number $x \in M$ either one of these two words is an infix of w_M :
 - $|J \xrightarrow{\perp} I \int \cdot \underbrace{\int I \xrightarrow{a} J | \dots \int I \xrightarrow{a} J |}_{x \text{ times}} \cdot \int I \xrightarrow{b} J |$, or
 - $\int I \xrightarrow{b} J | \cdot \underbrace{\int I \xrightarrow{a} J | \dots \int I \xrightarrow{a} J |}_{x \text{ times}} \cdot \int I \xrightarrow{b} J |$.

So by examination of the word w_M one can enumerate every number in the set M . Due to this result, we can establish that LOOP+ ω programs are capable of forming enumerators — algorithms for enumerating recursively enumerable sets.

5.3 Rice's Theorem for $\text{LOOP}+\omega$ Programs

In this section, we recall Rice's Theorem for WHILE programs and give an adaption of the theorem for $\text{LOOP}+\omega$ programs. Rice's Theorem is a powerful theorem proven in 1953 by Henry G. Rice for general recursive functions, however, here we will use the equivalent notion of WHILE programs. The theorem states that any semantic property of WHILE programs is not decidable. We shall first recall the notion of semantic properties and Rice's Theorem for the case of WHILE programs before we present the adaption for $\text{LOOP}+\omega$ programs:

We say that a function $f: \mathbb{N} \rightarrow \mathbb{N}$ is *WHILE-computable*, if there exists a WHILE program P such that for all $x \in \mathbb{N}$ it holds that $f(x) = \langle P \rangle(\sigma_x)(x_1)$. This means that the WHILE program P reads the argument x from the first variable x_0 and stores the result $f(x)$ in the variable x_1 after it finishes its calculations. A set of WHILE programs $A \subseteq \mathbb{W}$ then represents a semantic property if there exists a set of WHILE -computable functions \mathcal{F} such that the following holds for all WHILE programs $P \in \mathbb{W}$:

$$P \in A \iff P \text{ calculates a function in } \mathcal{F}$$

That means that membership of a program P to the set A is *solely* dependent on the function calculated by P . E.g. the set $A_{\text{square}} = \{P \in \mathbb{W} \mid P \text{ calculates the square of } x_0\}$ represents a semantic property. The associated set of functions is $\mathcal{F}_{\text{square}} = \{f: \mathbb{N} \rightarrow \mathbb{N} \mid f(x) = x^2\}$. On the other hand the set $A_{23} = \{P \in \mathbb{W} \mid P \text{ consists of 23 statements}\}$ does not represent a semantic property as there exist for instance programs in A_{23} which calculate a function in $\mathcal{F}_{\text{square}}$ but there exist also other programs which are not in A_{23} but still calculate a function in $\mathcal{F}_{\text{square}}$.

Rice's Theorem for WHILE programs now states that every set A which represents a non-trivial semantic property (non-trivial meaning $\emptyset \neq A \neq \mathbb{W}$) is undecidable, meaning that there exists no algorithm which can decide whether a WHILE program P is a member of A or not.

We will now transfer the concept of semantic properties from WHILE programs to $\text{LOOP}+\omega$ programs, defining semantic properties by means of the languages generated by the $\text{LOOP}+\omega$ programs:

Definition D.5.3 (SEMANTIC PROPERTIES OF $\text{LOOP}+\omega$ PROGRAMS):

A set of $\text{LOOP}+\omega$ programs $A \subseteq \mathbb{L}_\omega$ represents a **semantic property**, if there exists a set of $\text{LOOP}+\omega$ -generatable languages $\mathcal{L} \subseteq \mathcal{P}(\Sigma_{I/O}^\omega)$, such that for all $\text{LOOP}+\omega$ programs P it holds that

$$P \in A \iff \text{strip}(L(P)) \in \text{strip}(\mathcal{L}).$$

A set A which represents a semantic property is called **non-trivial**, if $\emptyset \neq A \neq \mathbb{L}_\omega$ holds.

With the above definition at hand, we can now just adapt Rice's Theorem to $\text{LOOP}+\omega$ programs, by simply replacing WHILE with $\text{LOOP}+\omega$ programs:

Theorem T.II (RICE'S THEOREM FOR $\text{LOOP}+\omega$ PROGRAMS):

Every set of $\text{LOOP}+\omega$ programs $A \subseteq \mathbb{L}_\omega$ which represents a non-trivial semantic property is undecidable.

Note that the strip function is indeed needed for the semantic properties. Otherwise the sets might become decidable: Assume we had defined for a semantic property that merely $P \in A \iff L(P) \in \mathcal{L}$ must hold. Then there existed semantic properties which are in fact decidable. Consider for instance the set

$$A = \left\{ P \in \mathbb{L}_\omega \mid \text{The first symbol of every word in } L(P) \text{ is } \langle I \xrightarrow{a} J \rangle \right\}.$$

The set A however, is actually decidable. For deciding whether $P \in A$ we only need to execute the ω -loop body of the program P once and examine which symbol is produced first. Recall that at least the symbol $\langle I \xrightarrow{a} J \rangle$ is output by P in every iteration of the ω -loop body, and therefore at least one symbol is generated. On the other hand, deciding whether the first symbol in $\text{strip}(L(P))$ is $\langle I \xrightarrow{a} J \rangle$ is undecidable due to Rice's Theorem for LOOP+ ω programs which we shall prove next:

Proof of Theorem T.II: The precondition for applying Theorem T.II is that the set $A \subseteq \mathbb{L}_\omega$ is non-trivial which means in particular that $A \neq \emptyset$ and thus there exists at least one program $P_a \in A$. Furthermore non-triviality also means that $A \neq \mathbb{L}_\omega$ and thus at least one different program $P_b \notin A$ exists. The programs P_a and P_b are of the following general forms (cf. Definition D.3.7):

$$\begin{aligned} P_a &= P'_a; \text{ LOOP } \omega \text{ DO } P''_a; \text{ pulse END} \\ P_b &= P'_b; \text{ LOOP } \omega \text{ DO } P''_b; \text{ pulse END,} \end{aligned}$$

where P'_a , P''_a , P'_b and P''_b are LOOP programs. Furthermore let \mathcal{L} be the set of languages associated to the semantic property which is represented by the set A , so

$$\mathcal{L} = \{L(P) \mid P \in A\}.$$

Due to the fact that A represents a semantic property it holds that $\text{strip}(L(P_a)) \in \text{strip}(\mathcal{L})$ and $\text{strip}(L(P_b)) \notin \text{strip}(\mathcal{L})$.

Assume now that the set A was decidable. Then there existed a computable algorithm \mathfrak{X}_A which decides A . We will now have to distinguish two cases: It either holds that $\langle \langle I \xrightarrow{a} J \rangle \rangle^\omega \notin \mathcal{L}$ (Case 1) or it holds that $\langle \langle I \xrightarrow{a} J \rangle \rangle^\omega \in \mathcal{L}$ (Case 2). Note that this is a complete distinction of cases, so for any set A representing a semantic property either Case 1 or Case 2 applies.

With \mathfrak{X}_A at hand, we now construct for Case 1 an algorithm \mathfrak{H}_1 , and for Case 2 an algorithm \mathfrak{H}_2 respectively, which both decide the Halting Problem for WHILE programs, using the algorithm \mathfrak{X}_A as a subprogram:

Case 1 ($\langle \langle I \xrightarrow{a} J \rangle \rangle^\omega \notin \mathcal{L}$):

Description of \mathfrak{H}_1 : Given a WHILE program P , where P is w.l.o.g. in Kleene Normal Form, i.e.

$$P = x_k := 1; \text{ WHILE } x_k \neq 0 \text{ DO } P' \text{ END,}$$

do the following:

1. Construct the $\mathcal{L}oop_\omega$ program Q_P^a as follows:

```

1:   $x_k := 1$ ;
2:   $x_{\text{init}} := 1$ ;
3:  LOOP  $\omega$  DO
4:      IF  $x_k \neq 0$  THEN
5:           $P'$ 
6:      ELSE
7:          IF  $x_{\text{init}} \neq 0$  THEN
8:               $P'_a$ ;
9:               $x_{\text{init}} := 0$ ;
10:         ELSE
11:              $P''_a$ ;
12:         END
13:     END;
14:     pulse
15: END

```

W.l.o.g. the variables used in P'_a and P''_a are disjoint from those used in P . Also, x_{init} does not occur in neither P'_a , P''_a nor P . Otherwise, rename variables first.

2. Pass Q_P^a to \mathfrak{X}_A .

3a. If $\mathfrak{X}_A(Q_P^a)$ returns “Yes” (i.e. $\text{strip}(\mathcal{L}(Q_P^a)) \in \text{strip}(\mathcal{L})$) then P halts, so return “Yes”.

3b. If $\mathfrak{X}_A(Q_P^a)$ returns “No” (i.e. $\text{strip}(\mathcal{L}(Q_P^a)) \notin \text{strip}(\mathcal{L})$) then P does not halt, so return “No”.

The program Q_P^a from the construction above behaves in the following manner: It first simulates the execution of P by iteratively executing P' , the while-loop body of P . After P terminates (if it ever terminates) the program P_a is simulated. For that the initialization code P'_a is executed and thereafter P''_a , the ω -loop body of P_a , is executed ad infinitum.

In case that P halts, the program Q_P^a generates the language $\{\underbrace{\quad}_I\}^n \cdot \mathcal{L}(P_a)$, where n is the number of times the while-loop body P' of P is executed before P halts. In case that P does not halt, the program Q_P^a generates the language $\{\underbrace{\quad}_I\}^\omega$.

Partial correctness of \mathfrak{H}_1 :

$$\begin{aligned}
P \text{ halts} &\implies \mathcal{L}(Q_P^a) = \{\underbrace{\quad}_I\}^n \cdot \mathcal{L}(P_a) \\
&\implies \text{strip}(\mathcal{L}(Q_P^a)) = \text{strip}(\{\underbrace{\quad}_I\}^n \cdot \mathcal{L}(P_a)) \\
&\implies \text{strip}(\mathcal{L}(Q_P^a)) = \text{strip}(\mathcal{L}(P_a)) && \text{(D.4.4)} \\
&\implies \text{strip}(\mathcal{L}(Q_P^a)) \in \text{strip}(\mathcal{L}) && (\text{strip}(\mathcal{L}(P_a)) \in \text{strip}(\mathcal{L})) \\
&\implies Q_P^a \in A && (A \text{ represents semantic property}) \\
&\implies \mathfrak{X}_A(Q_P^a) = \text{“Yes”} && (\mathfrak{X}_A \text{ decides } A) \\
&\implies \mathfrak{H}_1(P) = \text{“Yes”} && (\text{Step 3a of } \mathfrak{H}_1)
\end{aligned}$$

$$\begin{aligned}
P \text{ does not halt} &\implies L(Q_P^a) = \{\sim^I\}^\omega \\
&\implies \text{strip}(L(Q_P^a)) = \text{strip}(\{\sim^I\}^\omega) \\
&\implies \text{strip}(L(Q_P^a)) = \varepsilon
\end{aligned} \tag{D.4.4}$$

Observe that in Case 1 it holds that $\{\sim^I\}^\omega \notin \mathcal{L}$ which implies $\varepsilon \notin \text{strip}(\mathcal{L})$.

$$\begin{aligned}
&\implies \text{strip}(L(Q_P^a)) \notin \text{strip}(\mathcal{L}) && \text{(Case 1)} \\
&\implies Q_P^a \notin A && (A \text{ represents semantic property}) \\
&\implies \mathfrak{X}_A(Q_P^a) = \text{"No"} && (\mathfrak{X}_A \text{ decides } A) \\
&\implies \mathfrak{H}_1(P) = \text{"No"} && (\text{Step 3b of } \mathfrak{H}_1)
\end{aligned}$$

Total correctness of \mathfrak{H}_1 : All steps of \mathfrak{H}_1 which do not invoke \mathfrak{X}_A are obviously computable and terminate. By assumption the invocation of \mathfrak{X}_A is also computable and terminates. Hence \mathfrak{H}_1 is computable and terminates.

The algorithm \mathfrak{H}_1 is hence a correct algorithm which decides the Halting Problem in Case 1.

Case 2 ($\{\sim^I\}^\omega \in \mathcal{L}$):

Description of \mathfrak{H}_2 : Given a WHILE program P , where P is w.l.o.g. in Kleene Normal Form, i.e.

$$P = x_k := 1; \text{ WHILE } x_k \neq 0 \text{ DO } P' \text{ END,}$$

do the following:

1. Construct the LOOP+ ω program Q_P^b as follows:

```

1:   $x_k := 1;$ 
2:   $x_{\text{init}} := 1;$ 
3:  LOOP  $\omega$  DO
4:      IF  $x_k \neq 0$  THEN
5:           $P'$ 
6:      ELSE
7:          IF  $x_{\text{init}} \neq 0$  THEN
9:               $P'_b;$ 
10:              $x_{\text{init}} := 0;$ 
11:          ELSE
12:              $P''_b;$ 
13:          END
14:      END;
15:  pulse
16:  END

```

W.l.o.g. the variables used in P'_b and P''_b are disjoint from those used in P . Also, x_{init} does not occur in neither P'_b , P''_b nor P .

2. Pass Q_P^b to \mathfrak{X}_A .
- 3a. If $\mathfrak{X}_A(Q_P^b)$ returns “Yes” (i.e. $\text{strip}(L(Q_P^b)) \in \text{strip}(\mathcal{L})$) then $\text{strip}(L(Q_P^b))$ must be $\{\sim\}^\omega$ and therefore P does not halt, so return “No”.
- 3b. If $\mathfrak{X}_A(Q_P^b)$ returns “No” (i.e. $\text{strip}(L(Q_P^b)) \notin \text{strip}(\mathcal{L})$) then $L(Q_P^b)$ must be $L(P_b)$ and therefore P halts, so return “Yes”.

The program Q_P^b from the construction above behaves in the following manner: It first simulates the execution of P by iteratively executing P' , the while-loop body of P . After P terminates (if it ever terminates) the program P_b is simulated. For that the initialization code P'_b is executed and thereafter P''_b , the loop body of P_b is executed ad infinitum.

In case that P halts, the program Q_P^b generates the language $\{\sim\}^n \cdot L(P_b)$, where n is the number of times the while-loop body P' of P is executed before P halts. In case that P does not halt, the program Q_P^b generates the language $\{\sim\}^\omega$.

Partial correctness of \mathfrak{H}_2 :

$$\begin{aligned}
P \text{ halts} &\implies L(Q_P^b) = \{\sim\}^n \cdot L(P_b) \\
&\implies \text{strip}(L(Q_P^b)) = \text{strip}(\{\sim\}^n \cdot L(P_b)) \\
&\implies \text{strip}(L(Q_P^b)) = \text{strip}(L(P_b)) && \text{(D.4.4)} \\
&\implies \text{strip}(L(Q_P^b)) \notin \text{strip}(\mathcal{L}) && (\text{strip}(L(P_b)) \notin \text{strip}(\mathcal{L})) \\
&\implies Q_P^b \notin A && (A \text{ represents a semantic property}) \\
&\implies \mathfrak{X}_A(Q_P^b) = \text{“No”} && (\mathfrak{X}_A \text{ decides } A) \\
&\implies \mathfrak{H}_2(P) = \text{“Yes”} && (\text{Step 3b of } \mathfrak{H}_2)
\end{aligned}$$

$$\begin{aligned}
P \text{ does not halt} &\implies L(Q_P^b) = \{\sim\}^\omega \\
&\implies \text{strip}(L(Q_P^b)) = \text{strip}(\{\sim\}^\omega) \\
&\implies \text{strip}(L(Q_P^b)) = \varepsilon && \text{(D.4.4)}
\end{aligned}$$

Observe that in Case 2 it holds that $\{\sim\}^\omega \in \mathcal{L}$ which implies $\varepsilon \in \text{strip}(\mathcal{L})$.

$$\begin{aligned}
&\implies \text{strip}(L(Q_P^b)) \in \text{strip}(\mathcal{L}) && \text{(Case 2)} \\
&\implies Q_P^b \in A && (A \text{ represents a semantic property}) \\
&\implies \mathfrak{X}_A(Q_P^b) = \text{“Yes”} && (\mathfrak{X}_A \text{ decides } A) \\
&\implies \mathfrak{H}_2(P) = \text{“No”} && (\text{Step 3a of } \mathfrak{H}_2)
\end{aligned}$$

Total correctness of \mathfrak{H}_2 : All steps of \mathfrak{H}_2 which do not invoke \mathfrak{X}_A are obviously computable and terminate. By assumption the invocation of \mathfrak{X}_A is also computable and terminates. Hence \mathfrak{H}_2 is computable and terminates.

The algorithm \mathfrak{H}_2 is hence a correct algorithm which decides the Halting Problem in Case 2.

So for both Case 1 and Case 2 we have a correct algorithm which decides the Halting Problem. As we made a complete case distinction, there always exists an algorithm for deciding the Halting Problem if there exists an algorithm \mathfrak{X}_A which decides A . As this is a contradiction to the undecidability of the Halting Problem, such an algorithm \mathfrak{X}_A cannot exist and thus any semantic property A is undecidable. \square

5.4 The Word Problem

A very natural problem which comes to mind when dealing with formal languages is the Word Problem. Given some generating or accepting device \mathfrak{A} and a word w , the Word Problem is to give an answer to the question whether $w \in L(\mathfrak{A})$.

In the context of ω -languages, however, giving a finite description of the word w is already somewhat challenging, as w is an ω -word and therefore infinite. In order to give a finite description, one has to make use of concepts such as Büchi Automata, ω -pushdown automata or logical formulas. All of these concepts are, however, already capable of describing an entire ω -language rather than only a single ω -word.

Hence the Word Problem is not very meaningful in the context of ω -languages. More suitable problems for ω -languages which are related to the Word Problem are the Inclusion Problem or the Intersection Problem, which will be discussed later in this chapter.

5.5 The Emptiness Problem and the Intersection Problem

The Emptiness Problem is the first problem we study for LOOP+ ω programs. It is the question whether a given program generates any word at all. We first give a formal definition of the classical version of the Emptiness Problem:

Problem P.5.2 (CLASSICAL EMPTINESS PROBLEM FOR $Loop_\omega$):

Given: A LOOP+ ω program $P \in \mathbb{L}_\omega$.

Question: Does it hold that $L(P) = \emptyset$?

Even though the Emptiness Problem is undecidable for Turing Machines, it is in fact decidable for LOOP+ ω programs:

Corollary C.5.9 (DECIDABILITY OF THE CLASSIC. EMPTINESS PROB. FOR $Loop_\omega$):

The Classical Emptiness Problem for $Loop_\omega$ is decidable.

Proof: The proof of Corollary C.5.9 is very easy as the answer to the question whether $L(P) = \emptyset$ holds is always “No”: A syntactically correct LOOP+ ω program P has the form

$$P = P_1; \text{ LOOP } \omega \text{ DO } P_2; \text{ pulse END,}$$

where P_1 and P_2 are both LOOP programs. Thus every LOOP+ ω program P produces at least one word in which infinitely many times the symbol $\sim\sim\sim$ occurs. In other words, the intersection $L(P) \cap \{(\Sigma_{I/O}^* \cdot \sim\sim\sim)^\omega\}$ is never empty for any LOOP+ ω program P . Hence, $L(P)$ cannot be empty for any P , so the answer to Problem P.5.2 is “No” for any program P and therefore Problem P.5.2 is decidable. \square

As we can see, for LOOP+ ω programs the Classical Emptiness Problem is not very interesting. It gets, however, more interesting, if we take a look at the essence of the Emptiness Problem: The question is, whether an instance of a generating

algorithm produces some sort of output by means of its semantics. The pulse of the program, however, is more of a syntactic obligation: Every word of the language of any $\text{LOOP}+\omega$ program contains infinitely many \sim^I symbols, no matter what the program calculates. For that reason, we study a more meaningful variant of the Emptiness Problem — the Essential Emptiness Problem:

Problem P.5.3 (ESSENTIAL EMPTINESS PROBLEM FOR Loop_ω):

Given: A $\text{LOOP}+\omega$ program $P \in \mathbb{L}_\omega$.

Question: Does it hold that $L(P) \setminus \{\sim^I\}^\omega = \emptyset$?

Lemma L.5.2 (UNDECIDABILITY OF THE ESSENT. EMPTINESS PROB. FOR Loop_ω):

The Essential Emptiness Problem for Loop_ω is undecidable.

Proof: Assume the Essential Emptiness Problem was decidable for Loop_ω . Then there existed a computable algorithm \mathfrak{Em} which decides for any given $\text{LOOP}+\omega$ program P whether $L(P) \setminus \{\sim^I\}^\omega = \emptyset$. With \mathfrak{Em} at hand, we construct an algorithm \mathfrak{H} which decides the Halting Problem for WHILE programs, using the algorithm \mathfrak{Em} as a subprogram.

Description of \mathfrak{H} : Given a WHILE program P , where P is w.l.o.g. in Kleene Normal Form. Then do the following:

1. Calculate the $\text{LOOP}+\omega$ simulation O_P of P (cf. Definition D.5.2).
2. Pass O_P to \mathfrak{Em} .
- 3a. If $\mathfrak{Em}(O_P)$ returns “Yes” (i.e. $L(O_P) \setminus \{\sim^I\}^\omega = \emptyset$) then P does not halt, so return “No”.
- 3b. If $\mathfrak{Em}(O_P)$ returns “No” (i.e. $L(O_P) \setminus \{\sim^I\}^\omega \neq \emptyset$) then P halts, so return “Yes”.

Partial correctness of \mathfrak{H} :

$$\begin{aligned}
P \text{ halts} &\implies L(O_P) = \{\sim^I\}^n \cdot \left\{ \left[I \xrightarrow{a} J \mid \sim^I \right]^\omega \right\} && \text{(L.5.1)} \\
&\implies L(O_P) \setminus \{\sim^I\}^\omega \neq \emptyset \\
&\implies \mathfrak{Em}(O_P) = \text{“No”} && (\mathfrak{Em} \text{ decides Essent. Emptiness Probl.}) \\
&\implies \mathfrak{H}(P) = \text{“Yes”} && (\text{Step 3b of } \mathfrak{H})
\end{aligned}$$

$$\begin{aligned}
P \text{ does not halt} &\implies L(O_P) = \{\sim^I\}^\omega && \text{(L.5.1)} \\
&\implies L(O_P) \setminus \{\sim^I\}^\omega = \emptyset \\
&\implies \mathfrak{Em}(O_P) = \text{“Yes”} && (\mathfrak{Em} \text{ decides Essent. Emptiness Probl.}) \\
&\implies \mathfrak{H}(P) = \text{“No”} && (\text{Step 3a of } \mathfrak{H})
\end{aligned}$$

Total correctness of \mathfrak{H} : All steps of \mathfrak{H} which do not invoke \mathfrak{Em} are obviously computable and terminate. By assumption the invocation of \mathfrak{Em} is also computable and terminates. Hence \mathfrak{H} is computable and terminates.

The algorithm \mathfrak{H} is hence a correct algorithm which decides the Halting Problem, which is a contradiction to the undecidability of the Halting Problem. Therefore \mathfrak{Em} cannot exist and thus the Essential Emptiness Problem is undecidable. \square

A problem related to the Emptiness Problem is the question whether the intersection of the language of a LOOP+ ω program with some regular language is empty. We call this the Intersection Problem for $\mathcal{L}oop_\omega$:

Problem P.5.4 (INTERSECTION PROBLEM FOR $\mathcal{L}oop_\omega$):

Given: A LOOP+ ω program $P \in \mathbb{L}_\omega$ and a Büchi Automaton \mathfrak{A} .

Question: Does it hold that $L(P) \cap L(\mathfrak{A}) = \emptyset$?

This is a very important problem, since many properties regarding safety, reliability and liveness can be expressed as regular languages. For instance, all languages definable in Linear-Time Temporal Logic (LTL) are ω -regular. Unfortunately, this problem, too, is undecidable:

Lemma L.5.3 (UNDECIDABILITY OF THE INTERSECTION PROBLEM FOR $\mathcal{L}oop_\omega$):

The Intersection Problem for $\mathcal{L}oop_\omega$ is undecidable.

Proof: Assume the Intersection Problem was decidable for $\mathcal{L}oop_\omega$. Then there existed a computable algorithm \mathfrak{Int} which decides whether the intersection of the language of any given LOOP+ ω program with any given regular language is empty. With \mathfrak{Int} at hand, we construct an algorithm \mathfrak{H} which decides the Halting Problem for WHILE programs, using the algorithm \mathfrak{Int} as a subprogram.

Description of \mathfrak{H} : Given a WHILE program P , where P is w.l.o.g. in Kleene Normal Form. Then do the following:

1. Calculate the LOOP+ ω simulation O_P of P .
2. Pass O_P and \mathfrak{A} with $L(\mathfrak{A}) = \{\tilde{\omega}\}^\omega$ to \mathfrak{Int} .
- 3a. If $\mathfrak{Int}(O_P, \mathfrak{A})$ returns “Yes” (i.e. $L(O_P) \cap \{\tilde{\omega}\}^\omega = \emptyset$) then P halts, so return “Yes”.
- 3b. If $\mathfrak{Int}(O_P, \mathfrak{A})$ returns “No” (i.e. $L(O_P) \cap \{\tilde{\omega}\}^\omega \neq \emptyset$) then P does not halt, so return “No”.

Partial correctness of \mathfrak{H} :

$$\begin{aligned}
P \text{ halts} &\implies L(O_P) = \{\tilde{\omega}\}^n \cdot \left\{ \left[I \xrightarrow{a} J \mid \cdot \tilde{\omega} \right] \right\}^\omega && \text{(L.5.1)} \\
&\implies L(O_P) \cap \{\tilde{\omega}\}^\omega = \emptyset \\
&\implies L(O_P) \cap L(\mathfrak{A}) = \emptyset \\
&\implies \mathfrak{Int}(O_P, \mathfrak{A}) = \text{“Yes”} && (\mathfrak{Int} \text{ decides Intersection Problem}) \\
&\implies \mathfrak{H}(P) = \text{“Yes”} && (\text{Step 3a of } \mathfrak{H})
\end{aligned}$$

$$\begin{aligned}
P \text{ does not halt} &\implies L(O_P) = \{\tilde{\omega}\}^\omega && \text{(L.5.1)} \\
&\implies L(O_P) \cap \{\tilde{\omega}\}^\omega = \{\tilde{\omega}\}^\omega \neq \emptyset \\
&\implies L(O_P) \cap L(\mathfrak{A}) \neq \emptyset \\
&\implies \mathfrak{Int}(O_P, \mathfrak{A}) = \text{“No”} && (\mathfrak{Int} \text{ decides Intersection Problem}) \\
&\implies \mathfrak{H}(P) = \text{“No”} && (\text{Step 3b of } \mathfrak{H})
\end{aligned}$$

Total correctness of \mathfrak{H} : All steps of \mathfrak{H} which do not invoke \mathfrak{Int} are obviously computable and terminate. By assumption the invocation of \mathfrak{Int} is also computable and terminates. Hence \mathfrak{H} is computable and terminates.

The algorithm \mathfrak{H} is hence a correct algorithm which decides the Halting Problem, which is a contradiction to the undecidability of the Halting Problem. Therefore \mathfrak{Int} cannot exist and thus the Intersection Problem is undecidable. \square

5.6 The Equivalence and the Inclusion Problem

Another interesting problem is the so called Equivalence Problem. It deals with the question whether the languages of two $\text{LOOP}+\omega$ programs are equal. Formally, the problem is defined as follows:

Problem P.5.5 (THE EQUIVALENCE PROBLEM FOR Loop_ω):

Given: Two $\text{LOOP}+\omega$ programs $P_1, P_2 \in \mathbb{L}_\omega$.

Question: Does it hold that $L(P_1) = L(P_2)$?

Just like in the case of ordinary Turing Machines, the Equivalence Problem is undecidable for Loop_ω , too:

Lemma L.5.4 (UNDECIDABILITY OF THE EQUIVALENCE PROBLEM FOR Loop_ω):

The Equivalence Problem for Loop_ω is undecidable.

Proof: Assume the Equivalence Problem was decidable for Loop_ω . Then there existed a computable algorithm \mathfrak{Eq} which decides whether the languages of any two $\text{LOOP}+\omega$ programs are equivalent. With \mathfrak{Eq} at hand, we construct an algorithm \mathfrak{H} which decides the Halting Problem for WHILE programs, using the algorithm \mathfrak{Eq} as a subprogram.

Description of \mathfrak{H} : Given a WHILE program P , where P is w.l.o.g. in Kleene Normal Form. Then do the following:

1. Calculate the $\text{LOOP}+\omega$ simulation O_P of P .
2. Pass O_P and P' to \mathfrak{Eq} , where P' is given by:

`skip; LOOP ω DO skip; pulse END`

Note that $L(P') = \{\sim^l\}^\omega$.

- 3a. If $\mathfrak{Eq}(O_P, P')$ returns “Yes” (i.e. $L(O_P) = L(P')$) then P does not halt, so return “No”.
- 3b. If $\mathfrak{Eq}(O_P, P')$ returns “No” (i.e. $L(O_P) \neq L(P')$) then P halts, so return “Yes”.

Partial correctness of \mathfrak{H} :

$$\begin{aligned}
P \text{ halts} &\implies L(O_P) = \{\underbrace{\sim\sim\sim}_I\}^n \cdot \left\{ \left[I \xrightarrow{a} J \mid \cdot \underbrace{\sim\sim\sim}_I \right] \right\}^\omega & (\text{L.5.1}) \\
&\implies L(O_P) \neq \{\underbrace{\sim\sim\sim}_I\}^\omega \\
&\implies L(O_P) \neq L(P') \\
&\implies \mathfrak{E}q(O_P, P') = \text{“No”} & (\mathfrak{E}q \text{ decides Equivalence Problem}) \\
&\implies \mathfrak{H}(P) = \text{“Yes”} & (\text{Step 3b of } \mathfrak{H})
\end{aligned}$$

$$\begin{aligned}
P \text{ does not halt} &\implies L(O_P) = \{\underbrace{\sim\sim\sim}_I\}^\omega & (\text{L.5.1}) \\
&\implies L(O_P) = L(P') \\
&\implies \mathfrak{E}q(O_P, P') = \text{“Yes”} & (\mathfrak{E}q \text{ decides Equivalence Problem}) \\
&\implies \mathfrak{H}(P) = \text{“No”} & (\text{Step 3a of } \mathfrak{H})
\end{aligned}$$

Total correctness of \mathfrak{H} : All steps of \mathfrak{H} which do not invoke $\mathfrak{E}q$ are obviously computable and terminate. By assumption the invocation of $\mathfrak{E}q$ is also computable and terminates. Hence \mathfrak{H} is computable and terminates.

The algorithm \mathfrak{H} is hence a correct algorithm which decides the Halting Problem, which is a contradiction to the undecidability of the Halting Problem. Therefore $\mathfrak{E}q$ cannot exist and thus the Equivalence Problem is undecidable. \square

A problem closely related to the Equivalence Problem is the Inclusion Problem. The problem deals with the question of whether the language of a LOOP+ ω program is a subset of the language of another LOOP+ ω program:

Problem P.5.6 (THE INCLUSION PROBLEM FOR $\mathcal{L}oop_\omega$):

Given: Two LOOP+ ω programs $P_1, P_2 \in \mathbb{L}_\omega$.

Question: Does it hold that $L(P_1) \subseteq L(P_2)$?

The undecidability of the Equivalence Problem, however, not surprisingly entails the undecidability the Inclusion Problem:

Lemma L.5.5 (UNDECIDABILITY OF THE INCLUSION PROBLEM FOR $\mathcal{L}oop_\omega$):

The Inclusion Problem for $\mathcal{L}oop_\omega$ is undecidable.

Proof: Assume the Inclusion Problem was decidable for $\mathcal{L}oop_\omega$. Then there existed a computable algorithm \mathfrak{Inc} which decides whether the language of any given LOOP+ ω program is included in the language of any other given LOOP+ ω program. With \mathfrak{Inc} at hand, we construct an algorithm $\mathfrak{E}q$ which decides the Equivalence Problem for $\mathcal{L}oop_\omega$, using the algorithm \mathfrak{Inc} as a subprogram.

Description of $\mathfrak{E}q$: Given two LOOP+ ω programs P_1 and P_2 . Then do the following:

1. Pass P_1 and P_2 to \mathfrak{Inc} .
- 2a. If both $\mathfrak{Inc}(P_1, P_2)$ and $\mathfrak{Inc}(P_2, P_1)$ return “Yes” (i.e. both $L(P_1) \subseteq L(P_2)$ and $L(P_2) \subseteq L(P_1)$ and therefore $L(P_1) = L(P_2)$) then return “Yes”.
- 2b. If either $\mathfrak{Inc}(P_1, P_2)$ or $\mathfrak{Inc}(P_2, P_1)$ or both return “No” (i.e. either $L(P_1) \not\subseteq L(P_2)$ or $L(P_2) \not\subseteq L(P_1)$ or both and therefore $L(P_1) \neq L(P_2)$) then return “No”.

Partial correctness of \mathfrak{Eq} :

$$\begin{aligned}
P_1 = P_2 &\implies L(P_1) \subseteq L(P_2) \text{ and } L(P_2) \subseteq L(P_1) \\
&\implies \mathfrak{Inc}(P_1, P_2) = \text{“Yes”} && (\mathfrak{Inc} \text{ decides Inclusion Problem}) \\
&\quad \text{and } \mathfrak{Inc}(P_2, P_1) = \text{“Yes”} \\
&\implies \mathfrak{Eq}(P_1, P_2) = \text{“Yes”} && (\text{Step 2a of } \mathfrak{H}) \\
\\
P_1 \neq P_2 &\implies L(P_1) \not\subseteq L(P_2) \text{ or } L(P_2) \not\subseteq L(P_1) \\
&\implies \mathfrak{Inc}(P_1, P_2) = \text{“No”} && (\mathfrak{Inc} \text{ decides Inclusion Problem}) \\
&\quad \text{or } \mathfrak{Inc}(P_2, P_1) = \text{“No”} \\
&\implies \mathfrak{Eq}(P_1, P_2) = \text{“No”} && (\text{Step 2b of } \mathfrak{H})
\end{aligned}$$

Total correctness of \mathfrak{Eq} : All steps of \mathfrak{Eq} which do not invoke \mathfrak{Inc} are obviously computable and terminate. By assumption the invocation of \mathfrak{Inc} is also computable and terminates. Hence \mathfrak{Eq} is computable and terminates.

The algorithm \mathfrak{Eq} is hence a correct algorithm which decides the Equivalence Problem, which is a contradiction to the undecidability of the Equivalence Problem. Therefore \mathfrak{Inc} cannot exist and thus the Inclusion Problem is undecidable. \square

5.7 The Regularity Problem

Up until now we have studied classical decision problems for formal languages in the context of $\text{LOOP}+\omega$ -generated languages. In the remainder of this chapter, we study decision problems more specific to $\text{LOOP}+\omega$ languages. A very interesting property of a $\text{LOOP}+\omega$ program for instance is, whether the language generated by the program is ω -regular or not:

Problem P.5.7 (THE REGULARITY PROBLEM FOR Loop_ω):

Given: A $\text{LOOP}+\omega$ program $P \in \mathbb{L}_\omega$.

Question: Is $L(P)$ ω -regular?

In consideration of the fact that all aforementioned problems are undecidable, it is not surprising that this problem is undecidable as well:

Lemma L.5.6 (UNDECIDABILITY OF THE REGULARITY PROBLEM FOR Loop_ω):

The Regularity Problem for Loop_ω is undecidable.

When dealing with problems like the Regularity Problem, the real power of Rice’s Theorem for $\text{LOOP}+\omega$ programs (Theorem T.II) becomes evident. By means of Rice’s Theorem, we can give a proof of Lemma L.5.6 which is much simpler than proving undecidability by reducing the Halting Problem to the Regularity Problem. For a comparison, the much longer proof by reduction from the Halting Problem can be found in Appendix A.3.

Proof of Lemma L.5.6 using Rice's Theorem for LOOP+ ω Programs (T.II):

Let $A \subseteq \mathbb{L}_\omega$ be the set of LOOP+ ω programs that generate an ω -regular language, so $A = \{P \in \mathbb{W} \mid L(P) \in \mathcal{R}eg_\omega\}$. The set of languages associated with A is $\mathcal{R}eg_\omega$. We now have to show merely two things: (1) A represents indeed a semantic property and (2) A is non-trivial.

We start with showing that A represents a semantic property: For any WHILE program $P \in \mathbb{W}$ the following holds:

$$\begin{aligned} P \in A &\implies L(P) \in \mathcal{R}eg_\omega \\ &\implies \{\sim^I\}^n \cdot \text{strip}(L(P)) \in \mathcal{R}eg_\omega && \text{(C.4.7)} \\ &\implies \text{strip}(L(P)) \in \mathcal{R}eg_\omega && \text{(R.2.1)} \end{aligned}$$

$\text{strip}(L(P))$ has no leading pulse symbols so it holds that $\text{strip}(L(P)) \in \text{strip}(\mathcal{R}eg_\omega)$:

$$\implies \text{strip}(L(P)) \in \text{strip}(\mathcal{R}eg_\omega)$$

$$\begin{aligned} P \notin A &\implies L(P) \notin \mathcal{R}eg_\omega \\ &\implies \{\sim^I\}^n \cdot \text{strip}(L(P)) \notin \mathcal{R}eg_\omega && \text{(C.4.7)} \\ &\implies \text{strip}(L(P)) \notin \mathcal{R}eg_\omega && \text{(R.2.1)} \end{aligned}$$

$\text{strip}(L(P))$ has no leading pulse symbols so it holds that $\text{strip}(L(P)) \notin \text{strip}(\mathcal{R}eg_\omega)$:

$$\implies \text{strip}(L(P)) \notin \text{strip}(\mathcal{R}eg_\omega)$$

This concludes the proof that A does indeed represent a semantic property.

In order for A to be non-trivial, it remains to show that there are programs in A and programs outside of A : For instance the language of the program P_{in} given by

```

1: skip;
2: LOOP  $\omega$  DO
3:     skip;
4:     pulse
5: END

```

is $L(P_{\text{in}}) = \{\sim^I\}^\omega$, which is obviously ω -regular, whereas the language of the program P_{out} given by

```

1:  $x_0 := 1$ ;
2: LOOP  $\omega$  DO
3:     LOOP  $x_0$  DO
4:         sendJ a
5:     END;
5:      $x_0 := x_0 + 1$ ;
6:     pulse
7: END

```

is $L(P_{\text{out}}) = \prod_{i=1}^{\infty} \left\{ \left(I \xrightarrow{a} J \right)^i \cdot \underbrace{\quad}_I \right\}$, which is not ω -regular by an obvious pumping argument. For a more detailed elaboration on why a language of the form of $L(P_{\text{out}})$ is not ω -regular, please refer to the proof of Lemma L.A.5 in Appendix A.3 on page 104.

As A represents exactly the semantic property of a program's language being regular, deciding regularity for a program P happens to be exactly the same as deciding whether it holds that $P \in A$. As, however, Rice's Theorem for $\text{LOOP}+\omega$ programs applies to A , membership of A is undecidable and therefore the Regularity Problem is undecidable, too. \square

5.8 The Pulse–Distance Boundedness Problem

The next problem we study is a problem we call the Pulse–Distance Boundedness Problem. It deals with the question whether the distance of any two pulse symbols is bounded by a constant for every word in the language of a $\text{LOOP}+\omega$ program P . In order to formally define the problem, we first define the pulse–distance bound of a language:

Definition D.5.4 (PULSE–DISTANCE BOUND):

The pulse–distance bound $\text{pdb}(w) \in \mathbb{N} \cup \{\infty\}$ of an ω -word w is the maximum distance between any two pulse symbols in w . If this distance is not bounded by any constant, then $\text{pdb}(w) = \infty$.

The pulse–distance bound of a language L is defined as

$$\text{pdb}(L) = \max_{w \in L} \{\text{pdb}(w)\}$$

From Definition D.5.4 follows immediately the following corollary, which will be needed in an upcoming proof:

Corollary C.5.10:

Let L_1 be an ordinary language with $\text{pdb}(L_1) = c$ and let L_2 be an ω -language. Then the following holds for the pulse–distance bound $L_1 \cdot L_2$:

$$\text{pdb}(L_1 \cdot L_2) = \infty \iff \text{pdb}(L_2) = \infty$$

By means of Definition D.5.4, we can now formally define the Pulse–Distance Bound Problem:

Problem P.5.8 (THE PULSE–DISTANCE BOUNDEDNESS PROBLEM):

Given: A $\text{LOOP}+\omega$ program $P \in \mathbb{L}_\omega$.

Question: Is the number of symbols between any two pulse symbols bounded by some constant for every word in $L(P)$?

This, too, is a relevant problem, because if such a bound is found, this bound can aid in finding a more precise approximation of $L(P)$. Unfortunately, this problem is undecidable as well:

Corollary C.5.11 (UNDECIDABILITY OF THE PULSE–DISTANCE BOUNDEDNESS PROBLEM):

The Pulse–Distance Boundedness Problem is undecidable.

Proof of Corollary C.5.11: Again the proof is done via Rice’s Theorem for LOOP+ ω programs: Let $A \subseteq \mathbb{L}_\omega$ be the set of programs that generate a language in which the distance between any two symbols is bounded for every word. The set of languages associated with A is $\mathcal{L} = \{L \subseteq \Sigma_{I/O}^\omega \mid \text{pdb}(L) = c < \infty\}$. As stripping leading pulse symbols can obviously neither bound the pulse–distance of a pulse–distance–unbounded language nor can it remove the bound from a pulse–distance–bounded language, it holds for any language L that $L \in \mathcal{L} \iff \text{strip}(L) \in \text{strip}(\mathcal{L})$.

We now have to show two things: (1) A represents indeed a semantic property and (2) A is non–trivial. We start with showing that A represents a semantic property. For any WHILE program P the following holds:

$$\begin{aligned} P \in A &\implies \text{pdb}(L(P)) = c < \infty \\ &\implies \text{pdb}(\{\sim^I\}^n \cdot \text{strip}(L(P))) = c < \infty \end{aligned} \tag{C.4.7}$$

$$\implies \text{pdb}(\text{strip}(L(P))) = c < \infty \tag{C.5.10}$$

$$\implies \text{strip}(L(P)) \in \mathcal{L}$$

$$\implies \text{strip}(\text{strip}(L(P))) \in \text{strip}(\mathcal{L}) \quad (L \in \mathcal{L} \iff L \in \text{strip}(\mathcal{L}))$$

$$\implies \text{strip}(L(P)) \in \text{strip}(\mathcal{L}) \quad (\text{strip is idempotent})$$

$$\begin{aligned} P \notin A &\implies \text{pdb}(L(P)) = \infty \\ &\implies \text{pdb}(\{\sim^I\}^n \cdot \text{strip}(L(P))) = \infty \end{aligned} \tag{C.4.7}$$

$$\implies \text{pdb}(\text{strip}(L(P))) = \infty \tag{C.5.10}$$

$$\implies \text{strip}(L(P)) \notin \mathcal{L}$$

$$\implies \text{strip}(\text{strip}(L(P))) \notin \text{strip}(\mathcal{L}) \quad (L \in \mathcal{L} \iff L \in \text{strip}(\mathcal{L}))$$

$$\implies \text{strip}(L(P)) \notin \text{strip}(\mathcal{L}) \quad (\text{strip is idempotent})$$

This concludes the proof that A does indeed represent a semantic property.

It remains to show A is not trivial, which is given as there exist programs which generate a language in \mathcal{L} , but not every program generates a language in \mathcal{L} . Namely, the pulse–distance is obviously bounded by 0 for the language of the program P_{in} (cf. page 63), whereas the pulse–distance is obviously unbounded for the language of the program P_{out} (cf. page 63).

As A represents exactly the semantic property of a program’s language’s pulse–distance being bounded, deciding pulse–distance boundedness for a program P happens to be exactly the same as deciding whether it holds that $P \in A$. As, however, Rice’s Theorem for LOOP+ ω programs applies to A , membership of A is undecidable and therefore the Pulse–Distance Boundedness Problem is undecidable, too. \square

5.9 The ω -Power Problem

The last problem we examine is the problem whether the language of a $\text{LOOP}+\omega$ program is of the form $L_1 \cdot L_2^\omega$. We call this problem the ω -Power Problem and its formal definition reads as follows:

Problem P.5.9 (THE ω -POWER PROBLEM):

Given: A $\text{LOOP}+\omega$ program $P \in \mathbb{L}_\omega$.

Question: Is $L(P)$ of the form $L(P) = L_1 \cdot L_2^\omega$ for some two languages L_1, L_2 ?

As this problem also deals with semantic properties of programs, it is not surprising that this problem is undecidable, too:

Corollary C.5.12 (UNDECIDABILITY OF THE ω -POWER PROBLEM):

The ω -Power Problem is undecidable.

Proof: Let $A \subseteq \mathbb{L}_\omega$ be the set of programs that generate a language of the form $L(P) = L_1 \cdot L_2^\omega$ for some two languages L_1, L_2 . The set of languages associated with A is $\mathcal{L} = \{L \subseteq \Sigma_{1/O}^\omega \mid L \text{ is of the form } L = L_1 \cdot L_2^\omega \text{ for some two languages } L_1, L_2\}$.

We now have to show two things: (1) A represents indeed a semantic property and (2) A is non-trivial. We start with showing that A represents a semantic property. For any WHILE program P the following holds:

$$\begin{aligned}
 P \in A &\implies L(P) = L_1 \cdot L_2^\omega \\
 &\implies \{\text{wavy}\}^n \cdot \text{strip}(L(P)) = L_1 \cdot L_2^\omega && \text{(C.4.7)} \\
 &\implies \text{strip}(\{\text{wavy}\}^n \cdot \text{strip}(L(P))) = \text{strip}(L_1 \cdot L_2^\omega) \\
 &\implies \text{strip}(L(P)) = \text{strip}(L_1 \cdot L_2^\omega) && \text{(D.4.4)} \\
 &\implies \text{strip}(L(P)) \in \{\text{strip}(w) \mid w \in L_1 \cdot L_2^\omega\} \\
 &\implies \text{strip}(L(P)) \in \{\text{strip}(w) \mid w \in \mathcal{L}\} \\
 &\implies \text{strip}(L(P)) \in \text{strip}(\mathcal{L}) && \text{(D.4.4)}
 \end{aligned}$$

Now for the converse direction we show $\text{strip}(L(P)) \in \text{strip}(\mathcal{L}) \implies P \in A$:

$$\begin{aligned}
 &\text{strip}(L(P)) \in \text{strip}(\mathcal{L}) \\
 &\implies \exists L_1, L_2: \text{strip}(L(P)) = \text{strip}(L_1 \cdot L_2^\omega) \\
 &\implies \exists L_1, L_2: \text{strip}(\{\text{wavy}\}^n \cdot \text{strip}(L(P))) = \text{strip}(L_1 \cdot L_2^\omega) && \text{(C.4.7)}
 \end{aligned}$$

Note that $L(P)$ has a common prefix $\{\text{wavy}\}^n$ with a constant n . So in order for $\text{strip}(L(P)) = \text{strip}(L_1 \cdot L_2^\omega)$ to hold, $L_1 \cdot L_2^\omega$ must have the same longest common prefix $\{\text{wavy}\}^n$. Therefore we can find two languages L'_1 and L'_2 such that $L_1 \cdot L_2^\omega = \{\text{wavy}\}^n \cdot L'_1 \cdot L'_2{}^\omega$.

$$\implies \exists L'_1, L'_2: \text{strip}(\{\text{wavy}\}^n \cdot \text{strip}(L(P))) = \text{strip}(\{\text{wavy}\}^n \cdot L'_1 \cdot L'_2{}^\omega)$$

As $\text{strip}(\{\text{wavy}\}^n \cdot \text{strip}(L(P))) = \text{strip}(L(P))$ and $L'_1 \cdot L'_2{}^\omega = \text{strip}(L(P))$ it follows that $\text{strip}(L'_1 \cdot L'_2{}^\omega) = L'_1 \cdot L'_2{}^\omega$:

$$\implies \exists L'_1, L'_2: \text{strip}(L(P)) = L'_1 \cdot L'_2{}^\omega$$

$$\begin{aligned}
&\implies \exists L'_1, L'_2: \{\sim^I\}^n \cdot \text{strip}(L(P)) = \{\sim^I\}^n \cdot L'_1 \cdot L'^{\omega}_2 \\
&\implies \exists L'_1, L'_2: L(P) = \{\sim^I\}^n \cdot L'_1 \cdot L'^{\omega}_2 \\
&\implies \exists L''_1, L'_2: L(P) = L''_1 \cdot L'^{\omega}_2 \\
&\implies P \in A
\end{aligned} \tag{C.4.7}$$

This concludes the proof that A does indeed represent a semantic property.

It remains to show A is not trivial, which is given as there exist programs which generate a language in \mathcal{L} , but not every program generates a language in \mathcal{L} . Namely, the language of P_{in} (cf. page 63) is obviously of the demanded form, whereas the language of P_{out} (cf. page 63) is obviously not of the demanded form.

As A represents exactly the semantic property that the language of a program is of the form $L_1 \cdot L_2^\omega$, deciding the ω -Power Problem for a program P happens to be exactly the same as deciding whether it holds that $P \in A$. As, however, Rice's Theorem for LOOP+ ω programs applies to A , membership of A is undecidable and therefore the ω -Power Problem is undecidable, too. \square

In conclusion we can establish that virtually every interesting property of the language of a LOOP+ ω program is undecidable. One way of dealing with undecidable problems is to try and decide such problems anyway, although this can of course only be achieved for certain subsets of all LOOP+ ω programs.

The way we will cope with the obstacle of undecidability is to find an ω -regular over-approximation of the language of a LOOP+ ω program, because for the class of ω -regular languages all of the above decision problems are in fact decidable. We will study how to compute such an over-approximation in the next chapter.

6

Over-Approximation of LOOP+ ω -Generated Languages

As we have learned in the previous chapters, many decision problems concerning the class of LOOP+ ω -generated languages, which are relevant for the verification of LOOP+ ω programs, are undecidable. It is therefore desirable to over-approximate the language generated by a LOOP+ ω program in a way, such that certain decision problems become decidable.

One way of doing so is to over-approximate the language of a LOOP+ ω program in such a way that the over-approximation is ω -regular. Such an ω -regular over-approximation is on one hand quite favorable, as many decision problems are decidable for these languages and the languages can even be intersected. On the other hand, an ω -regular over-approximation might be too coarse.

In the following we present a basic method on how to obtain an ω -regular language which over-approximates the actual language generated by a LOOP+ ω program and prove the correctness of this method.

6.1 ω -Regular Over-Approximation

The structure of LOOP+ ω programs allows for a very intuitive ω -regular over-approximation of the language generated by a LOOP+ ω program. Because of its intuitivity, we call this over-approximation the *natural* over-approximation. It is defined as follows:

Definition D.6.1 (NATURAL OVER-APPROXIMATION):

The natural over-approximation $\widehat{L}: (\mathbb{L} \cup \mathbb{L}_\omega) \rightarrow \mathcal{P}(\Sigma_{I/O}^\omega)$ of the language generated by a LOOP or LOOP+ ω program is given as follows:

$$\begin{aligned}
\widehat{L}(\text{skip}) &= \{\varepsilon\} \\
\widehat{L}(x_i := x_j + c) &= \{\varepsilon\} \\
\widehat{L}(\text{send}_J \ a) &= \{?I \xrightarrow{a} J|\} \\
\widehat{L}(\text{send}_J \ b) &= \{?I \xrightarrow{b} J|\} \\
\widehat{L}(\text{pulse}) &= \{\sim\sim\sim\} \\
\widehat{L}(\text{RECV}_J \ a \ \text{DO} \ P_1 \\
&\quad \text{OR} \ b \ \text{DO} \ P_2 \\
&\quad \text{OR} \ _ \ \text{DO} \ P_3 \ \text{END}) &= \{|J \xrightarrow{a} I|\} \cdot \widehat{L}(P_1) \cup \{|J \xrightarrow{b} I|\} \cdot \widehat{L}(P_2) \\
&\quad \cup \{|J \xrightarrow{_} I|\} \cdot \widehat{L}(P_3) \\
\widehat{L}(P_1; P_2) &= \widehat{L}(P_1) \cdot \widehat{L}(P_2) \\
\widehat{L}(\text{LOOP} \ x \ \text{DO} \ P \ \text{END}) &= \widehat{L}(P)^* \\
\widehat{L}(\text{LOOP} \ \omega \ \text{DO} \ P \ \text{END}) &= \widehat{L}(P)^\omega
\end{aligned}$$

We explain the definition above in more detail: For every non-composed statement, its language is naturally over-approximated by the I/O-symbol it produces in the I/O-language.

Message reception is naturally over-approximated by the union over each receive-symbol to which accordingly the natural over-approximation of the language generated by the program, which is executed on receiving that symbol, is concatenated.

The natural over-approximation of the concatenation of two programs is given by the concatenation of the two natural over-approximations of the programs.

Given that for LOOP+ ω programs it is generally not decidable how many times the loop body of a finite loop is iterated in the course of the execution, we simply over-approximate the loop by any finite number of loop iterations. This is achieved by taking the Kleene-closure of the natural over-approximation of the loop body. Analogously, the ω -loop body is executed infinitely many times, so the ω -loop is naturally over-approximated by the ω -closure of the natural over-approximation of the loop body.

Obviously the definition of the natural over-approximation can be employed as an algorithm and is hence directly applicable to the analysis of LOOP+ ω programs. In order to show that the natural over-approximation is in fact a *correct and ω -regular over-approximation* of the language generated by a LOOP+ ω program, we need to prove the following theorem:

Theorem T.III (CORRECTNESS OF THE NATURAL OVER-APPROXIMATION):

Let P be any LOOP program, let P_ω be any LOOP+ ω program and let $S_\varepsilon \in \mathbb{S}$ be any set of configurations for which $L(S_\varepsilon) = \{\varepsilon\}$ holds. Then the following holds:

- (I) The natural over-approximation $\widehat{L}(P)$ is a correct regular over-approximation of the language $L_{S_\varepsilon}(P)$ generated by P , i.e.

$$L_{S_\varepsilon}(P) \subseteq \widehat{L}(P) \in \mathcal{Reg}$$

- (II) The natural over-approximation $\widehat{L}(P_\omega)$ is a correct ω -regular over-approximation of the language $L_{S_\varepsilon}(P_\omega)$ generated by P_ω , i.e.

$$L_{S_\varepsilon}(P_\omega) \subseteq \widehat{L}(P_\omega) \in \mathcal{Reg}_\omega$$

Even though the natural over-approximation appears quite intuitive, proving its correctness is not trivial. In the next part of this chapter we prove Theorem T.III.

Proof of Theorem T.III

For proving Theorem T.III we need to show two things: (1) $\widehat{L}(P)$ is a correct over-approximation of the language $L_{S_\varepsilon}(P)$ and (2) $\widehat{L}(P)$ is ω -regular (or regular if P is a LOOP program). We will show the former first:

Lemma L.6.1:

Let P be any LOOP+ ω or LOOP program and let S_ε be any set of configurations for which $L(S_\varepsilon) = \{\varepsilon\}$ holds. Then it holds that $L_{S_\varepsilon}(P) \subseteq \widehat{L}(P)$.

Proof: We prove Lemma L.6.1, again by structural induction over all LOOP+ ω programs, very much like the proof of Lemma L.4.1. For the induction basis, we have the non-composed statements. As the induction hypothesis we assume that Lemma L.6.1 holds for programs P_1 , P_2 and P_3 . We then show that, given the induction hypothesis, Lemma L.6.1 also holds for the statements composed of P_1 , P_2 and P_3 .

Induction Basis

For the induction basis, we prove Theorem T.III for every non-composed statement:

Empty Statement

$$L_{S_\varepsilon}(\text{skip}) = L(S_\varepsilon) \tag{L.4.3}$$

$$= \{\varepsilon\}$$

$$= \widehat{L}(\text{skip}) \tag{D.6.1}$$

Assignment

$$L_{S_\varepsilon}(x_i := x_j + c) = L(S_\varepsilon) \tag{L.4.3}$$

$$= \{\varepsilon\}$$

$$= \widehat{L}(x_i := x_j + c) \tag{D.6.1}$$

Message Sending and Pulsing

$$\begin{aligned}
L_{S_\varepsilon}(\text{send}_J \ a) &= L(S_\varepsilon) \cdot \{I \xrightarrow{a} J\} & (L.4.3) \\
&= \{\varepsilon\} \cdot \{I \xrightarrow{a} J\} \\
&= \{I \xrightarrow{a} J\} \\
&= \widehat{L}(\text{send}_J \ a) & (D.6.1)
\end{aligned}$$

The case for `sendJ b` and `pulse` is completely analogous.

We have now shown that for all non-composed statements Theorem T.III holds which concludes the proof of the induction basis. Furthermore, note that for all programs P consisting of only a single non-composed statement the natural over-approximation is not really an over-approximation but rather $L(P) = \widehat{L}(P)$ holds.

Induction Hypothesis

As our induction hypothesis we assume that Theorem T.III holds for the LOOP programs P_1 , P_2 and P_3 . More formally we assume $L(P_1) \subseteq \widehat{L}(P_1)$, $L(P_2) \subseteq \widehat{L}(P_2)$ and $L(P_3) \subseteq \widehat{L}(P_3)$.

Induction Step

For the induction step it remains to show that, given the induction hypothesis, Theorem T.III holds for the composed statements as well:

Message Reception

$$\begin{aligned}
&L_{S_\varepsilon}(\text{RECV}_J \ a \ \text{DO} \ P_1 \ \text{b} \ \text{DO} \ P_2 \ _ \ \text{DO} \ P_3 \ \text{END}) \\
&= L_{S_\varepsilon \cdot |J \xrightarrow{a} I\} (P_1) \cup L_{S_\varepsilon \cdot |J \xrightarrow{b} I\} (P_2) \cup L_{S_\varepsilon \cdot |J \xrightarrow{\perp} I\} (P_3) & (L.4.3) \\
&= L_{|J \xrightarrow{a} I\} \cdot L_{S_\varepsilon} (P_1) \cup L_{|J \xrightarrow{b} I\} \cdot L_{S_\varepsilon} (P_2) \cup L_{|J \xrightarrow{\perp} I\} \cdot L_{S_\varepsilon} (P_3) & (C.4.6) \\
&= |J \xrightarrow{a} I\} \cdot L_{S_\varepsilon} (P_1) \cup |J \xrightarrow{b} I\} \cdot L_{S_\varepsilon} (P_2) \cup |J \xrightarrow{\perp} I\} \cdot L_{S_\varepsilon} (P_3) & (C.4.5) \\
&\subseteq |J \xrightarrow{a} I\} \cdot \widehat{L}(P_1) \cup |J \xrightarrow{b} I\} \cdot \widehat{L}(P_2) \cup |J \xrightarrow{\perp} I\} \cdot \widehat{L}(P_3) & (I.H.) \\
&= \widehat{L}(\text{RECV}_J \ a \ \text{DO} \ P_1 \ \text{OR} \ \text{b} \ \text{DO} \ P_2 \ \text{OR} \ _ \ \text{DO} \ P_3 \ \text{END}) & (D.6.1)
\end{aligned}$$

This concludes the proof for the case of message reception. Note that the over-approximation does *not* arise through the definition of the natural over-approximation of the receive statement, but rather by the induction hypothesis, i.e. in the case that $L_{S_\varepsilon}(P_1) = \widehat{L}_{S_\varepsilon}(P_1)$, $L_{S_\varepsilon}(P_2) = \widehat{L}_{S_\varepsilon}(P_2)$ and $L_{S_\varepsilon}(P_3) = \widehat{L}_{S_\varepsilon}(P_3)$ it holds that

$$\begin{aligned}
&L_{S_\varepsilon}(\text{RECV}_J \ a \ \text{DO} \ P_1 \ \text{b} \ \text{DO} \ P_2 \ _ \ \text{DO} \ P_3 \ \text{END}) \\
&= \widehat{L}(\text{RECV}_J \ a \ \text{DO} \ P_1 \ \text{OR} \ \text{b} \ \text{DO} \ P_2 \ \text{OR} \ _ \ \text{DO} \ P_3 \ \text{END}) .
\end{aligned}$$

Concatenation

$$\begin{aligned} & L_{S_\varepsilon}(P_1; P_2) \\ &= \bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)} \left(v \cdot L_{\{(\tau, \varepsilon)\}}(P_2) \right) \end{aligned} \quad (\text{L.4.3})$$

At this point, we do a short digression on unions of products in order to further simplify the expression above.

Digression: Unions of Products

Let v_1, \dots, v_k be words over the alphabet Σ and let L_1, \dots, L_k be languages over Σ . Now consider the language

$$\mathcal{L}_1 = \bigcup_{i \in \{1, \dots, k\}} \left(v_i \cdot L_i \right).$$

All the words in the language \mathcal{L}_1 are of the form $v_i \cdot z_i$ for $z_i \in L_i$. Every word of the form $v_i \cdot z_i$ for $z_i \in L_i$ is obviously also in the language

$$\mathcal{L}_2 = \left(\bigcup_{i \in \{1, \dots, k\}} \{v_i\} \right) \cdot \left(\bigcup_{i \in \{1, \dots, k\}} L_i \right).$$

If every word which is in \mathcal{L}_1 is also in \mathcal{L}_2 we can state that $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

According to the digression above, we can go on with our conclusion as follows:

$$\begin{aligned} & \bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)} \left(v \cdot L_{\{(\tau, \varepsilon)\}}(P_2) \right) \\ & \subseteq \left(\bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)} \{v\} \right) \cdot \left(\bigcup_{(\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)} L_{\{(\tau, \varepsilon)\}}(P_2) \right) \quad (\text{Digression}) \\ & = L_{S_\varepsilon}(P_1) \cdot L_{\{(\tau, \varepsilon) \mid (\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)\}}(P_2) \quad (\text{D.3.10, C.4.4}) \end{aligned}$$

By assumption S_ε satisfies $L(S_\varepsilon) = \{\varepsilon\}$. In addition $\{(\tau, \varepsilon) \mid (\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)\}$ obviously also satisfies $L(\{(\tau, \varepsilon) \mid (\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)\}) = \{\varepsilon\}$. We may therefore apply the induction hypothesis to both $L_{S_\varepsilon}(P_1)$ and $L_{\{(\tau, \varepsilon) \mid (\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)\}}(P_2)$:

$$\begin{aligned} & \subseteq \widehat{L}(P_1) \cdot \widehat{L}(P_2) \quad (\text{I.H.}) \\ & = \widehat{L}(P_1; P_2) \quad (\text{D.6.1}) \end{aligned}$$

This concludes the proof for the concatenation. As we can see, there are two over-approximations here: The latter obviously arises from the induction hypothesis. This is the over-approximation of the languages of the two programs P_1 and P_2 . The former over-approximation, however, arises from concatenating the over-approximations of P_1 and P_2 . We basically lose the information which words that

emerge from executing P_1 are prolonged by P_2 in which way. Instead, we just concatenate *every* possible prolongation by P_2 to *every* word emerging from P_1 .

Example E.6.1 (NATURAL OVER-APPROXIMATION OF THE CONCAT.):

Consider the following LOOP program:

```

1:  RECVJ
2:      a DO  $x_1 := x_2 + 1$ ;
3:      OR b DO  $x_1 := x_2 + 1$ ;
4:      OR _ DO skip
5:  END;
6:  LOOP  $x_1$  DO
7:      sendJ a
8:  END

```

This program attempts to read a symbol from the channel $C_{J \rightarrow I}$. If it is successful, it sends symbol a to channel $C_{I \rightarrow J}$.

The natural over-approximation of this program is given by:

$$\begin{aligned}
& \widehat{\text{L}}(\text{RECV}_J [\dots] \text{END}; \text{LOOP } x_1 \text{ DO } [\dots] \text{END}) \\
&= \widehat{\text{L}}(\text{RECV}_J [\dots] \text{END}) \cdot \widehat{\text{L}}(\text{LOOP } x_1 \text{ DO } [\dots] \text{END}) \\
&= \left(\left\{ |J \xrightarrow{a} I \right\} \cdot \varepsilon \right) \cup \left\{ |J \xrightarrow{b} I \right\} \cdot \varepsilon \cup \left\{ |J \xrightarrow{\perp} I \right\} \cdot \varepsilon \Big) \\
&\quad \cdot \widehat{\text{L}}(\text{LOOP } x_1 \text{ DO send}_J \text{ a END}) \\
&= \left(\left\{ |J \xrightarrow{a} I \right\} \right) \cup \left\{ |J \xrightarrow{b} I \right\} \cup \left\{ |J \xrightarrow{\perp} I \right\} \Big) \\
&\quad \cdot \widehat{\text{L}}(\text{LOOP } x_1 \text{ DO send}_J \text{ a END}) \\
&= \left\{ |J \xrightarrow{a} I \right\}, |J \xrightarrow{b} I \right\}, |J \xrightarrow{\perp} I \right\} \\
&\quad \cdot \widehat{\text{L}}(\text{LOOP } x_1 \text{ DO send}_J \text{ a END}) \\
&= \left\{ |J \xrightarrow{a} I \right\}, |J \xrightarrow{b} I \right\}, |J \xrightarrow{\perp} I \right\} \cdot \widehat{\text{L}}(\text{send}_J \text{ a})^* \\
&= \left\{ |J \xrightarrow{a} I \right\}, |J \xrightarrow{b} I \right\}, |J \xrightarrow{\perp} I \right\} \cdot \left\{ \langle I \xrightarrow{a} J \rangle \right\}^*
\end{aligned}$$

The actual language of this program, however, is

$$\left\{ |J \xrightarrow{a} I \right\}, |J \xrightarrow{b} I \right\} \cdot \left\{ \langle I \xrightarrow{a} J \rangle \right\} \cup \left\{ |J \xrightarrow{\perp} I \right\}.$$

Loops

$$\begin{aligned} & L_{S_\varepsilon}(\text{LOOP } x \text{ DO } P \text{ END}) \\ &= \{w \mid (\sigma, w) \in \llbracket \text{LOOP } x \text{ DO } P \text{ END} \rrbracket(S_\varepsilon)\} \end{aligned} \quad (\text{D.3.10})$$

$$= \left\{ w \mid (\sigma, w) \in \bigcup_{(\tau, \varepsilon) \in S_\varepsilon} \llbracket P \rrbracket^{\tau(x)}(\{(\tau, \varepsilon)\}) \right\} \quad (\text{D.3.8})$$

The expression $\llbracket P \rrbracket^{\tau(x)}(\{(\tau, \varepsilon)\})$ represents a $\tau(x)$ -fold application of P to $\{(\tau, \varepsilon)\}$ which is the same as $\underbrace{\llbracket P; \dots; P \rrbracket}_{\tau(x) \text{ times}}(\{(\tau, \varepsilon)\})$, so:

$$= \left\{ w \mid (\sigma, w) \in \bigcup_{(\tau, \varepsilon) \in S_\varepsilon} \underbrace{\llbracket P; \dots; P \rrbracket}_{\tau(x) \text{ times}}(\{(\tau, \varepsilon)\}) \right\}$$

Now, we over-approximate the number of loop iterations by allowing *any* number of iterations:

$$\begin{aligned} & \subseteq \left\{ w \mid (\sigma, w) \in \bigcup_{i \in \mathbb{N}} \underbrace{\llbracket P; \dots; P \rrbracket}_{i \text{ times}}(\{(\tau, \varepsilon)\}) \right\} \\ &= \bigcup_{i \in \mathbb{N}} \left\{ w \mid (\sigma, w) \in \underbrace{\llbracket P; \dots; P \rrbracket}_{i \text{ times}}(\{(\tau, \varepsilon)\}) \right\} \\ &= \bigcup_{i \in \mathbb{N}} L_{\{(\tau, \varepsilon)\}} \underbrace{(P; \dots; P)}_{i \text{ times}} \end{aligned} \quad (\text{D.3.10})$$

Since $\{(\tau, \varepsilon)\}$ obviously satisfies $L(\{(\tau, \varepsilon)\}) = \{\varepsilon\}$, and $P; \dots; P$ is a concatenation of programs, we may now apply the case for the concatenation to obtain:

$$\begin{aligned} & \subseteq \bigcup_{i \in \mathbb{N}} \widehat{L} \underbrace{(P; \dots; P)}_{i \text{ times}} \\ &= \bigcup_{i \in \mathbb{N}} \underbrace{\widehat{L}(P) \cdots \widehat{L}(P)}_{i \text{ times}} \quad (\text{D.6.1}) \\ &= \bigcup_{i \in \mathbb{N}} \widehat{L}(P)^i \\ &= \widehat{L}(P)^* \\ &= \widehat{L}(\text{LOOP } x \text{ DO } P \text{ END}) \end{aligned} \quad (\text{D.6.1})$$

 ω -Loops

$$\begin{aligned} & L_{S_\varepsilon}(\text{LOOP } \omega \text{ DO } P \text{ END}) \\ &= \{w \mid (\sigma, w) \in \llbracket \text{LOOP } \omega \text{ DO } P \text{ END} \rrbracket(\{(\sigma_0, \varepsilon)\})\} \end{aligned} \quad (\text{D.3.10})$$

$$= \left\{ w \mid (\sigma, w) \in \left\{ (\perp, v) \mid \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \llbracket P \rrbracket^k(\{(\sigma_0, \varepsilon)\}) \end{array} \right\} \right\} \quad (\text{D.3.8})$$

$$= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \llbracket P \rrbracket^k(\{(\sigma_0, \varepsilon)\}) \end{array} \right. \right\}$$

Since v_0 is inevitably equal to ε , we may simplify this to:

$$\begin{aligned} &= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_1 \cdots v_k) \in \llbracket P \rrbracket^k(\{(\sigma_0, \varepsilon)\}) \end{array} \right. \right\} \\ &= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_1 \cdots v_k) \in \underbrace{\llbracket P; \dots; P \rrbracket}_{k \text{ times}}(\{(\sigma_0, \varepsilon)\}) \end{array} \right. \right\} \\ &= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N}, \\ \text{such that } v_1 \cdots v_k \in L_{S_\varepsilon} \left(\underbrace{P; \dots; P}_{k \text{ times}} \right) \end{array} \right. \right\} \tag{D.3.10} \\ &\subseteq \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N}, \\ \text{such that } v_1 \cdots v_k \in \widehat{L} \left(\underbrace{P; \dots; P}_{k \text{ times}} \right) \end{array} \right. \right\} \tag{cf. Concatenation} \\ &= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N}, \\ \text{such that } v_1 \cdots v_k \in \underbrace{\widehat{L}(P) \cdots \widehat{L}(P)}_{k \text{ times}} \end{array} \right. \right\} \tag{D.6.1} \\ &= \widehat{L}(P)^\omega \\ &= \widehat{L}(\text{LOOP } \omega \text{ DO } P \text{ END}) \tag{D.6.1} \end{aligned}$$

This concludes the Proof of Lemma L.6.1. \square

It remains to show that the natural over-approximation of a LOOP program is regular and the natural over-approximation of a LOOP+ ω program is ω -regular. We show the former first:

Lemma L.6.2 (REGULARITY OF NATURAL OVER-APPROXIMATION OF LOOP PROGRAMS):

For any LOOP Program $P \in \mathbb{L}$ the natural over-approximation $\widehat{L}(P)$ of the language $L(P)$ generated by P is regular.

Proof: The proof of Lemma L.6.2 is again done by structural induction over all LOOP programs. For the induction basis, we have the non-composed statements. As the induction hypothesis we assume that Lemma L.6.2 holds for programs P_1 , P_2 and P_3 . We then show that, given the induction hypothesis, Lemma L.6.2 also holds for the statements composed of P_1 , P_2 and P_3 .

Induction Basis

For the non-composed statements it holds that their natural over-approximation consists of either the language $\{\varepsilon\}$ or some language $\{a\}$ for $a \in \Sigma_{I/O}$. Both are obviously regular.

Induction Hypothesis

As our induction hypothesis we assume that Lemma L.6.2 holds for the LOOP programs P_1 , P_2 and P_3 . More formally we assume that $\widehat{L}(P_1)$, $\widehat{L}(P_2)$ and $\widehat{L}(P_3)$ are all regular.

Induction Step

We now prove that, given the induction hypothesis, Lemma L.6.2 holds for every composed statement:

- Given that $\widehat{L}(P_1)$, $\widehat{L}(P_2)$ and $\widehat{L}(P_3)$ are all regular, the natural over-approximation of the receive statement is a union over concatenations of regular languages which is again regular (cf. Remark R.2.1).
- Given that $\widehat{L}(P_1)$ and $\widehat{L}(P_2)$ are both regular, the natural over-approximation of the concatenation of P_1 and P_2 is a concatenation of regular languages which is again regular (cf. Remark R.2.1).
- Given that $\widehat{L}(P_1)$ is regular, the natural over-approximation of the finite loop is a Kleene-closure of a regular language which is again regular (cf. Remark R.2.1). □

With the aid of Lemma L.6.2 we can prove the ω -regularity of the natural over-approximation of a LOOP+ ω program quite easily:

Lemma L.6.3 (ω -REGULARITY OF THE NATURAL OVER-APPROXIMATION):
For any LOOP+ ω Program $P \in \mathbb{L}_\omega$ the natural over-approximation $\widehat{L}(P)$ of the language $L(P)$ generated by P is ω -regular.

Proof: A LOOP+ ω program basically has the structure

$$P_1; \text{ LOOP } \omega \text{ DO } P_2 \text{ END}$$

where P_1 and P_2 are both LOOP programs. The natural over-approximation of any LOOP+ ω program is thus given by

$$\widehat{L}(P_1) \cdot \widehat{L}(P_1)^\omega.$$

By means of Lemma L.6.2 both $\widehat{L}(P_1)$ and $\widehat{L}(P_2)$ are regular because P_1 and P_2 are LOOP programs. Since $\widehat{L}(P_1)$ and $\widehat{L}(P_2)$ are regular we can conclude by Remark R.2.10 that $\widehat{L}(P_1) \cdot \widehat{L}(P_1)^\omega$ is ω -regular. □

6.2 Remarks on ω -Context-Sensitive Over-Approximations

Sometimes an ω -regular over-approximation of a systems behavior might be too coarse. There are, however, two major difficulties when trying to refine the over-approximation to an ω -context-free one:

The first difficulty becomes apparent, when analyzing a whole *network* of communicating LOOP+ ω programs. For that, the languages of the participating LOOP+ ω programs must be synchronized. This operation is very much related to intersection and we will show in a later chapter that LOOP+ ω generated languages are not closed under synchronization. One way to circumvent this second difficulty would be to over-approximate only *one* of the LOOP+ ω programs by an ω -context-free language and all other programs by an ω -regular language. This would not be problematic when synchronizing as ω -context-free languages are closed under intersection with ω -regular languages.

The second difficulty is the difference in the information is stored in LOOP+ ω programs and in pushdown systems: LOOP+ ω programs use program variables to store information. Pushdown systems use a stack. Variables basically allow for random access to information for LOOP+ ω programs whereas the information stored in stacks can only be accessed in a last-in-first-out (LIFO) manner. Although LOOP+ ω programs can emulate a stack behavior as random access is more powerful than LIFO access, it is very difficult to automatically assert that a LOOP+ ω program does in fact exhibit a stack behavior using its variables. Even trying to come up with a LOOP+ ω program which internally simulates a stack is not so trivial: Even though it is possible, it results in quite a complicated program, because the stack content as well as the order of the content has to be coded into a finite number of program variables employing number theoretic tricks like the uniqueness of the prime factorization or the like.

7

Analysis of Networks of Communicating LOOP+ ω Programs

Up until now, we have only described the communication behavior and the over-approximation of the communication behavior of a single program from a local point of view. In this chapter, we will discuss how to analyze an entire network of communicating LOOP+ ω programs. For that, we will have to achieve three things:

1. Combine all local languages of the LOOP+ ω programs in a network into a single language which contains all interleavings of the local behaviors.
2. Ensure that only interleavings that obey the logics of channel system are allowed, for instance that a program P_I can read a symbol a from the input channel of program P_J if and only if the program P_J has sent a to program P_I prior to that.
3. Optionally, we may demand some fairness among the programs.

A language which captures the above aspects acts as an observer which observes all communication events in the network not from a local but from a global point of view, which takes the whole network into account. We will call such a language the *aether* of a network of N communicating LOOP+ ω programs P_1, \dots, P_N and we will denote this language by $\mathfrak{Aether}(\{P_1, \dots, P_N\})$. The calculation of the \mathfrak{Aether} -function will be achieved by means of an operation we call *synchronization of languages*, which we shall study first.

7.1 Synchronization of Languages

The general notion of the synchronization of two languages L_1 and L_2 with *possibly unequal* alphabets is supposed to be a hybrid between the interleaving and the intersection of L_1 and L_2 . It shall behave like an interleaving for symbols which

are in the alphabet of L_1 but not of L_2 and vice versa, and it shall behave like an intersection for symbols which are in the alphabet of both languages. We formally define the synchronization of two languages in the following and give a justification of this definition later in this section.

Definition D.7.1 (SYNCHRONIZATION OF LANGUAGES):

Let w be a word over the alphabet Σ and let Σ' be another alphabet. Then the **projection $w|_{\Sigma'}$ of the word w on the alphabet Σ'** is a monoid homomorphism from $(\Sigma, \cdot, \varepsilon)$ to $(\Sigma', \cdot, \varepsilon)$ which is defined as

$$a|_{\Sigma'} = \begin{cases} \varepsilon, & \text{if } a \notin \Sigma' \\ a & \text{if } a \in \Sigma' \end{cases}$$

Let $L_1 \subseteq \Sigma_1^\infty$ and $L_2 \subseteq \Sigma_2^\infty$ be two languages whose alphabets might differ. Then the **synchronization $L_1 \otimes L_2$ of the two languages L_1 and L_2** is defined as

$$L_1 \otimes L_2 = \{w \in (\Sigma_1 \cup \Sigma_2)^\infty \mid w|_{\Sigma_1} \in L_1 \text{ and } w|_{\Sigma_2} \in L_2\},$$

Recall that, since the projection is a monoid homomorphism $\cdot|_{\Sigma'} : \Sigma^* \rightarrow \Sigma'^*$, we only have to define the value of $\cdot|_{\Sigma'}$ for every $a \in \Sigma$ and not for every $w \in \Sigma^*$, since $a \cdot v|_{\Sigma'} = a|_{\Sigma'} \cdot v|_{\Sigma'}$ and $\varepsilon|_{\Sigma'} = \varepsilon$.

As we can see, the synchronization of two languages L_1 and L_2 is only defined by means of the property it has to satisfy. The definition contains no hint on how to actually construct such a synchronization. This is due to the fact that the two languages are likely to be effectively given by some sort of automaton or grammar and are thus regular, context-free or the like. However, we will learn later that not every class of languages is closed under synchronization. Therefore we will have to provide for every class of languages its own tailor-made *implementation* of the synchronization operator.

We will start by providing the synchronization operator for ordinary regular languages. It is in fact given by means of the well-known synchronized product for finite automata. This justifies the reasonability of Definition D.7.1. The synchronized product for finite automata is used to synchronize two automata which operate on overlapping alphabets. When processing a word, the synchronized product of the automata \mathfrak{A} and \mathfrak{B} takes a step for the automaton \mathfrak{A} if the currently read symbol is in the alphabet of \mathfrak{A} , and a step for \mathfrak{B} if the symbol is in the alphabet of \mathfrak{B} . If the next read symbol is in both alphabets, both automata must take a step. Formally, the synchronized product is defined as follows:

Definition D.7.2 (SYNCHRONIZED PRODUCT OF FINITE AUTOMATA):

Let $\mathfrak{A} = (Q_A, \Sigma_A, \delta_A, q_{0A}, F_A)$ and $\mathfrak{B} = (Q_B, \Sigma_B, \delta_B, q_{0B}, F_B)$ be two deterministic finite automata. Then the synchronized product of $\mathfrak{A} \boxtimes \mathfrak{B}$ of the two automata \mathfrak{A} and \mathfrak{B} is defined as

$$\mathfrak{A} \boxtimes \mathfrak{B} = (Q_A \times Q_B, \Sigma_A \cup \Sigma_B, \delta, (q_{0A}, q_{0B}), F_A \times F_B),$$

where

$$\delta((q_A, q_B), a) = \begin{cases} (\delta_A(q_A), \delta_B(q_B)), & \text{if } a \in \Sigma_A \cap \Sigma_B \\ (\delta_A(q_A), q_B), & \text{if } a \in \Sigma_A \setminus \Sigma_B \\ (q_A, \delta_B(q_B)), & \text{if } a \in \Sigma_B \setminus \Sigma_A. \end{cases}$$

The synchronized product of finite automata is a widely used operation and we will now justify Definition D.7.1 by proving that — in the case of regular languages — the synchronization $L_A \otimes L_B$ of two regular languages effectively given by the DFA \mathfrak{A} and \mathfrak{B} , respectively, is equivalent to the language $L(\mathfrak{A} \boxtimes \mathfrak{B})$ of the synchronized product of \mathfrak{A} and \mathfrak{B} :

Theorem T.IV (EQUIVALENCE OF THE SYNCHRONIZED PRODUCT AND THE SYNCHRONIZATION OF LANGUAGES FOR THE CASE OF REGULAR LANGUAGES):
Let $L_A \subseteq \Sigma_1^$ and $L_B \subseteq \Sigma_2^*$ be two regular languages effectively given by the DFA \mathfrak{A} and \mathfrak{B} . Then the language of the synchronized product $\mathfrak{A} \boxtimes \mathfrak{B}$ of \mathfrak{A} and \mathfrak{B} is equal to the synchronization $L_A \otimes L_B$ of the languages L_A and L_B , i.e. the following equality holds:*

$$L(\mathfrak{A} \boxtimes \mathfrak{B}) = L_A \otimes L_B$$

Proof of $L(\mathfrak{A} \boxtimes \mathfrak{B}) \subseteq L_A \otimes L_B$: Assume $w \in L(\mathfrak{A} \boxtimes \mathfrak{B})$. It obviously holds that $w \in (\Sigma_A \cup \Sigma_B)^*$.

Furthermore the run $\rho_w = (q_{(1)_A}, q_{(1)_B})(q_{(2)_A}, q_{(2)_B}) \cdots (q_{(|w|+1)_A}, q_{(|w|+1)_B})$ of $\mathfrak{A} \boxtimes \mathfrak{B}$ on w is accepting. The run of \mathfrak{A} on $w|_{\Sigma_A}$ is embedded in ρ_w : For any two consecutive states $(q_{(i)_A}, q_{(i)_B})$ and $(q_{(i+1)_A}, q_{(i+1)_B})$ in ρ_w the state of \mathfrak{A} may only change (i.e. an actual state transition from $q_{(i)_A}$ to $q_{(i+1)_A}$ is performed), if the symbol $w[i+1] \in \Sigma_A$. For $w[i+1] \notin \Sigma_A$ the state is simply repeated. The run ρ_w thus ends in a state $(q_{(|w|+1)_A}, q_{(|w|+1)_B})$ where $q_{(|w|+1)_A} \in F_A$ if and only if the the automaton \mathfrak{A} accepts $w|_{\Sigma_A}$, i.e. if and only if $w|_{\Sigma_A} \in L_A$.

The proof that $w|_{\Sigma_B} \in L_B$ is completely analogous and thus $w \in L_A \otimes L_B$ by Definition D.7.1.

Proof of $L(\mathfrak{A} \boxtimes \mathfrak{B}) \supseteq L_A \otimes L_B$: Assume $w \notin L(\mathfrak{A} \boxtimes \mathfrak{B})$. There are two cases: Either $w \notin (\Sigma_A \cup \Sigma_B)^*$ then obviously either \mathfrak{A} or \mathfrak{B} or both cannot accept $w|_{\Sigma_A}$ or $w|_{\Sigma_B}$, respectively, since at least one of the automata cannot perform a proper transition for at least one of the symbols of w and thus $w|_{\Sigma_A} \notin L_A$ or $w|_{\Sigma_B} \notin L_B$ or both and therefore $w \notin L_A \otimes L_B$.

The other case is that $w \in (\Sigma_A \cup \Sigma_B)^*$: Then the run $\rho_w = (q_{(1)_A}, q_{(1)_B})(q_{(2)_A}, q_{(2)_B}) \cdots (q_{(|w|+1)_A}, q_{(|w|+1)_B})$ of $\mathfrak{A} \boxtimes \mathfrak{B}$ on w is not accepting. That means that either $q_{(|w|+1)_A} \notin F_A$ or $q_{(|w|+1)_B} \notin F_B$ or both.

Assume $q_{(|w|+1)_A} \notin F_A$. The run of \mathfrak{A} on $w|_{\Sigma_A}$ is embedded in ρ_w : For any two consecutive states $(q_{(i)_A}, q_{(i)_B})$ and $(q_{(i+1)_A}, q_{(i+1)_B})$ in ρ_w the state of \mathfrak{A} may only change (i.e. an actual state transition from $q_{(i)_A}$ to $q_{(i+1)_A}$ is performed), if the symbol $w[i+1] \in \Sigma_A$. For $w[i+1] \notin \Sigma_A$ the state is simply repeated. The run ρ_w thus ends in a state $(q_{(|w|+1)_A}, q_{(|w|+1)_B})$ where $q_{(|w|+1)_A} \notin F_A$ if and only if the the automaton \mathfrak{A} rejects $w|_{\Sigma_A}$, i.e. if and only if $w|_{\Sigma_A} \notin L_A$.

The proof for $q_{(|w|+1)_B} \notin F_B$ and therefore $w|_{\Sigma_B} \notin L_B$ is completely analogous. Since either $w|_{\Sigma_A} \notin L_A$ or $w|_{\Sigma_B} \notin L_B$ holds, by Definition D.7.1 it holds that $w \notin L_A \otimes L_B$. \square

7.1.1 Synchronization of ω -regular Languages

Now that we have justified that Definition D.7.1 is in fact a reasonable definition for defining the interleaving of two languages whose alphabets are potentially unequal but overlapping, we will provide a synchronization operator for ω -regular languages.

Unfortunately, applying the ordinary synchronized product construction for DFA from Definition D.7.2 directly to two Büchi Automata \mathfrak{A} and \mathfrak{B} results in an unwanted behavior of the product automaton. This is because in this construction the set of final states is the product set of the final states of \mathfrak{A} and \mathfrak{B} . This means however, that the synchronized product $\mathfrak{A} \boxtimes \mathfrak{B}$ (when used as a Büchi Automaton) accepts an ω -word only if \mathfrak{A} and \mathfrak{B} would reach a final state at the same time. If now for some ω -word w the automaton \mathfrak{A} would for instance reach a final state on every even symbol of w and \mathfrak{B} would do so on every uneven symbol of w , then both \mathfrak{A} and \mathfrak{B} would have accepted w , but $\mathfrak{A} \boxtimes \mathfrak{B}$ would have rejected w , because this automaton never reaches a state (p, q) where *both* p and q are final states.

The solution is to construct a Büchi Automaton with *sets of final states* where every set has to be visited infinitely often in order to accept a word. Such automata are known as *Generalized Büchi Automata*:

Definition D.7.3 (GENERALIZED BÜCHI AUTOMATA):

A **Generalized Büchi Automaton (GBA)** \mathfrak{A} is a quintuple $\mathfrak{A} = (Q, \Sigma, \Delta, q_0, \mathcal{F})$ defined just like a NBA, except that it has a set of sets of final states $\mathcal{F} = \{F_1, \dots, F_k\}$ instead of a single set of final states F .

Runs of \mathfrak{A} on a word w is defined just like for ordinary NBA, but for GBA a run ρ_w of \mathfrak{A} on a word w is an **accepting run**, if every of the final state sets in \mathcal{F} is visited infinitely often, i.e. there exist $M_1, \dots, M_k \subseteq \mathbb{N}$ such that for every M_i it holds that $|M_i| = \infty$ and for every M_i it holds that $\forall j \in M_i: \rho_w[j] \in F_i$.

Just as with NBA, a word w is accepted, if there exists an accepting run of \mathfrak{A} on w . The language $L(\mathfrak{A})$ of the automaton \mathfrak{A} is defined as usual by $L(\mathfrak{A}) = \{w \in \Sigma^\omega \mid \mathfrak{A} \text{ accepts } w\}$.

We will now use these GBA to provide a construction for the synchronization of ω -regular languages:

Definition D.7.4 (SYNCHRONIZED PRODUCT FOR BÜCHI AUTOMATA):

Let $\mathfrak{A} = (Q_A, \Sigma_A, \Delta_A, q_{0A}, F_A)$ and $\mathfrak{B} = (Q_B, \Sigma_B, \Delta_B, q_{0B}, F_B)$ be two Büchi Automata and let \cdot . Then the synchronized product $\mathfrak{A} \boxtimes_\omega \mathfrak{B}$ of the two Büchi Automata \mathfrak{A} and \mathfrak{B} is defined by means of a GBA as

$$\mathfrak{A} \boxtimes_\omega \mathfrak{B} = (Q, \Sigma, \Delta, q_0, \mathcal{F}),$$

where $(Q, \Sigma, \Delta, q_0, F) = \mathfrak{A} \boxtimes \mathfrak{B}$ is the standard synchronized product construction from Definition D.7.2 (but applied to Büchi Automata) and $\mathcal{F} = \{F_A \times Q_B, Q_A \times F_B\}$.

We now have to prove that for the class of ω -regular languages the definition of \boxtimes_ω agrees with the general definition of synchronization. So we have to prove the following lemma:

Lemma L.7.1:

Let $L_A \subseteq \Sigma_1^\omega$ and $L_B \subseteq \Sigma_2^\omega$ be two ω -regular languages effectively given by the Büchi Automata \mathfrak{A} and \mathfrak{B} , respectively. Then the language $L(\mathfrak{A} \boxtimes_\omega \mathfrak{B})$ of the synchronized product $\mathfrak{A} \boxtimes_\omega \mathfrak{B}$ of \mathfrak{A} and \mathfrak{B} is equal to the synchronization $L_A \otimes L_B$ of the languages L_A and L_B , i.e. the following equality holds:

$$L(\mathfrak{A} \boxtimes_\omega \mathfrak{B}) = L_A \otimes L_B$$

Proof of $L(\mathfrak{A} \boxtimes_\omega \mathfrak{B}) \subseteq L_A \otimes L_B$: Assume $w \in L(\mathfrak{A} \boxtimes_\omega \mathfrak{B})$. Then it obviously holds that $w \in (\Sigma_A \cup \Sigma_B)^*$.

Furthermore the run $\rho_w = (q_{(1)_A}, q_{(1)_B})(q_{(2)_A}, q_{(2)_B})(q_{(3)_A}, q_{(3)_B}) \cdots$ of $\mathfrak{A} \boxtimes_\omega \mathfrak{B}$ on w is accepting which means in particular that the run contains infinitely many states (q_A, q_B) where $q_A \in F_A$. The run of \mathfrak{A} on $w|_{\Sigma_A}$ is embedded in ρ_w : For any two consecutive states $(q_{(i)_A}, q_{(i)_B})$ and $(q_{(i+1)_A}, q_{(i+1)_B})$ in ρ_w the state of \mathfrak{A} may only change (i.e. an actual state transition from $q_{(i)_A}$ to $q_{(i+1)_A}$ is performed), if the symbol $w[i+1] \in \Sigma_A$. For $w[i+1] \notin \Sigma_A$ the state is simply repeated. The run ρ_w thus contains infinitely many states (q_A, q_B) where $q_A \in F_A$ if and only if the automaton \mathfrak{A} accepts $w|_{\Sigma_A}$, i.e. if and only if $w|_{\Sigma_A} \in L_A$.

The proof that $w|_{\Sigma_B} \in L_B$ is completely analogous and thus $w \in L_A \otimes L_B$ by Definition D.7.1.

Proof of $L(\mathfrak{A} \boxtimes_\omega \mathfrak{B}) \supseteq L_A \otimes L_B$: Assume $w \notin L(\mathfrak{A} \boxtimes_\omega \mathfrak{B})$. There are two cases: Either $w \notin (\Sigma_A \cup \Sigma_B)^\omega$ then obviously either \mathfrak{A} or \mathfrak{B} or both cannot accept $w|_{\Sigma_A}$ or $w|_{\Sigma_B}$, respectively, since at least one of the automata cannot perform a proper transition for at least one of the symbols of w and thus $w|_{\Sigma_A} \notin L_A$ or $w|_{\Sigma_B} \notin L_B$ or both and therefore $w \notin L_A \otimes L_B$.

The other case is that $w \in (\Sigma_A \cup \Sigma_B)^\omega$: Then the run $\rho_w = (q_{(1)_A}, q_{(1)_B})(q_{(2)_A}, q_{(2)_B})(q_{(3)_A}, q_{(3)_B}) \cdots$ of $\mathfrak{A} \boxtimes_\omega \mathfrak{B}$ on w is not accepting. That means that either ρ_w does not contain infinitely many states (q_A, q_B) where $q_A \in F_A$ or ρ_w does not contain infinitely many states (q_A, q_B) where $q_B \in F_B$ or both.

Assume ρ_w does not contain infinitely many states (q_A, q_B) where $q_A \in F_A$. The run of \mathfrak{A} on $w|_{\Sigma_A}$ is embedded in ρ_w : For any two consecutive states $(q_{(i)_A}, q_{(i)_B})$ and $(q_{(i+1)_A}, q_{(i+1)_B})$ in ρ_w the state of \mathfrak{A} may only change (i.e. an actual state transition from $q_{(i)_A}$ to $q_{(i+1)_A}$ is performed), if the symbol $w[i+1] \in \Sigma_A$. For $w[i+1] \notin \Sigma_A$ the state is simply repeated. The run ρ_w thus does not contain infinitely many states (q_A, q_B) where $q_A \in F_A$ if and only if the automaton \mathfrak{A} rejects $w|_{\Sigma_A}$, i.e. if and only if $w|_{\Sigma_A} \notin L_A$.

The proof for the case that ρ_w does not contain infinitely many states (q_A, q_B) where $q_B \in F_B$ and therefore $w|_{\Sigma_B} \notin L_B$ is completely analogous. Since either $w|_{\Sigma_A} \notin L_A$ or $w|_{\Sigma_B} \notin L_B$ holds, by Definition D.7.1 it holds that $w \notin L_A \otimes L_B$. \square

We have provided a correct construction of the synchronization of ω -regular languages by means of GBA. As Moshe Ya'akov Vardi and Pierre Wolper showed in [VW84], the languages accepted by GBA are precisely the ω -regular languages. This means, that the synchronization of two ω -regular languages using GBA is again ω -regular, and therefore we can establish the following result:

Theorem T.V (CLOSURE ω -REGULAR LANGUAGES UNDER SYNCHRONIZATION):
Reg $_{\omega}$ is closed under synchronization.

7.1.2 Synchronization of ω -Context-Free Languages

The class of ω -context-free languages is more expressive than the class of ω -regular languages while the emptiness problem is still decidable (cf. Remark R.2.14). It would therefore be desirable to over-approximate a LOOP+ ω program's behavior by some ω -context-free language. However, as we will establish next, the class of ω -context-free languages is *not* closed under synchronization and therefore an ω -context-free over-approximation of the communication behavior of a LOOP+ ω program is not admissible for our analysis.

The reason that the class of ω -context-free languages is not closed under synchronization is that this class is not closed under intersection. We will deduce the close relationship between synchronization and intersection in the following. First, we note that in the case of equal alphabets, intersection and synchronization are actually equivalent:

Corollary C.7.13 (EQUIVALENCE OF SYNCHRONIZATION AND INTERSECTION FOR THE CASE OF EQUAL ALPHABETS):

Let $L_1 \subseteq \Sigma^\infty$ and $L_2 \subseteq \Sigma^\infty$ be two languages whose alphabets do not differ. Then the synchronization of L_1 and L_2 is equivalent to the intersection of L_1 and L_2 , i.e. the following equality holds:

$$L_1 \otimes L_2 = L_1 \cap L_2$$

Proof: Let $L_1 \subseteq \Sigma^\infty$ and $L_2 \subseteq \Sigma^\infty$ be two languages whose alphabets do not differ. Now we conclude:

$$\begin{aligned} L_1 \otimes L_2 &= \{w \in (\Sigma \cup \Sigma)^\infty \mid w|_{\Sigma} \in L_1 \text{ and } w|_{\Sigma} \in L_2\} & (D.7.1) \\ &= \{w \in \Sigma^\infty \mid w|_{\Sigma} \in L_1 \text{ and } w|_{\Sigma} \in L_2\} \\ &= \{w \in \Sigma^\infty \mid w \in L_1 \text{ and } w \in L_2\} & (w \in \Sigma^\infty \implies w|_{\Sigma} = w) \\ &= \{w \in \Sigma^\infty \mid w \in L_1\} \cap \{w \in \Sigma^\infty \mid w \in L_2\} \\ &= L_1 \cap L_2 \end{aligned}$$

□

We will use this result to show that, given a very simple supplementary condition, closure of a class of languages under synchronization implies closure under intersection:

Lemma L.7.2:

Let \mathcal{L} be a family of languages which is closed under addition and deletion of prefixes, i.e. $\forall w: L \in \mathcal{L} \iff \{w\} \cdot L \in \mathcal{L}$. Then \mathcal{L} is closed under intersection, if \mathcal{L} is closed under synchronization, i.e. for any two languages $L_1, L_2 \in \mathcal{L}$ it holds that

$$(L_1 \otimes L_2 \in \mathcal{L}) \implies (L_1 \cap L_2 \in \mathcal{L})$$

Proof: Let \mathcal{L} be a family of languages which is closed under addition and deletion of prefixes and synchronization. Furthermore let $L_1 \subseteq \Sigma_1^\infty$ and $L_2 \subseteq \Sigma_2^\infty$ be any two languages in \mathcal{L} . We then construct some word $w \in (\Sigma_1 \cup \Sigma_2)^*$ which contains every symbol from Σ_1 and every symbol from Σ_2 at least once. Then the languages $\{w\} \cdot L_1$ and $\{w\} \cdot L_2$ are languages over the same alphabet, namely $(\Sigma_1 \cup \Sigma_2)$. Also, since \mathcal{L} is closed under addition of prefixes, both $\{w\} \cdot L_1$ and $\{w\} \cdot L_2$ are in \mathcal{L} . Additionally, as \mathcal{L} is closed under synchronization, the language $(\{w\} \cdot L_1) \otimes (\{w\} \cdot L_2)$ is also in \mathcal{L} . We then conclude:

$$\begin{aligned} & (\{w\} \cdot L_1) \otimes (\{w\} \cdot L_2) \in \mathcal{L} \\ \implies & (\{w\} \cdot L_1) \cap (\{w\} \cdot L_2) \in \mathcal{L} && \text{(same alphabet, C.7.13 applies)} \\ \implies & \{w\} \cdot (L_1 \cap L_2) \in \mathcal{L} \\ \implies & L_1 \cap L_2 \in \mathcal{L} && (\mathcal{L} \text{ closed under prefix deletion}) \end{aligned}$$

□

We can now use this result to establish the final result that the class of ω -context-free languages is not closed under synchronization:

Theorem T.VI:

\mathcal{CFL}_ω is not closed under synchronization.

Proof: According to [CG77a] every ω -context-free language is a finite union of languages of the form $V \cdot W^\omega$, where V and W are ordinary context-free languages. As the language $\{w\}$ is obviously context-free and context-free languages are closed under concatenation (cf. Remark R.2.4), the language $\{w\} \cdot V$ is also context-free and therefore $\{w\} \cdot V \cdot W^\omega$ is ω -context-free and hence ω -context-free languages are closed under prefix addition and deletion. Therefore \mathcal{CFL}_ω satisfies the precondition of Lemma L.7.2.

Now assume \mathcal{CFL}_ω was closed under synchronization. Then, according to Lemma L.7.2, \mathcal{CFL}_ω is closed under intersection, too. Closure of \mathcal{CFL}_ω under intersection however, is a contradiction to Remark R.2.13 and therefore \mathcal{CFL}_ω cannot be closed under synchronization. □

7.2 Interleaving Languages of Communicating Programs

We will use the synchronization of languages as in Definition D.7.1 to generate an interleaving of all $\text{LOOP}+\omega$ programs participating in the network which we want to analyze as a first step in constructing the aether language. As we have learned in the previous section, the class of languages which is tractable for this method is the class of ω -regular languages. By means of the natural over-approximation as in Definition D.6.1 we have a correct ω -regular over-approximation of the language of a $\text{LOOP}+\omega$ program. In order to obtain the interleaving of the languages of a network of N communicating $\text{LOOP}+\omega$ programs P_1, \dots, P_N , we can therefore just take the

synchronization of all natural over-approximations of the languages of P_1, \dots, P_N , so we have

$$\text{Interleaving}(\{P_1, \dots, P_N\}) = \bigotimes_{I \in \{1, \dots, N\}} \left(\widehat{L}(P_I) \right).$$

The order in which we synchronize is not relevant as the synchronization of languages is always associative and commutative:

Lemma L.7.3 (PROPERTIES OF THE SYNCHRONIZED PRODUCT):

(I) \otimes is associative, i.e. for any three languages L_1, L_2 and L_3 the following equality holds:

$$(L_1 \otimes L_2) \otimes L_3 = L_1 \otimes (L_2 \otimes L_3)$$

(II) \otimes is commutative, i.e. for any two languages L_1 and L_2 the following equality holds:

$$L_1 \otimes L_2 = L_2 \otimes L_1$$

Proof of (I):

$$\begin{aligned} & (L_1 \otimes L_2) \otimes L_3 \\ = & \{w \in ((\Sigma_1 \cup \Sigma_2) \cup \Sigma_3)^\infty \mid w|_{\Sigma_1 \cup \Sigma_2} \in L_1 \otimes L_2 \text{ and } w|_{\Sigma_3} \in L_3\} \\ = & \{w \in ((\Sigma_1 \cup \Sigma_2) \cup \Sigma_3)^\infty \mid [w|_{\Sigma_1} \in L_1 \text{ and } w|_{\Sigma_2} \in L_2] \text{ and } w|_{\Sigma_3} \in L_3\} \\ = & \{w \in (\Sigma_1 \cup (\Sigma_2 \cup \Sigma_3))^\infty \mid w|_{\Sigma_1} \in L_1 \text{ and } [w|_{\Sigma_2} \in L_2 \text{ and } w|_{\Sigma_3} \in L_3]\} \\ = & \{w \in ((\Sigma_1 \cup \Sigma_2) \cup \Sigma_3)^\infty \mid w|_{\Sigma_1} \in L_1 \text{ and } w|_{\Sigma_2 \cup \Sigma_3} \in L_2 \otimes L_3\} \\ = & L_1 \otimes (L_2 \otimes L_3) \end{aligned}$$

Proof of (II)

$$\begin{aligned} & L_1 \otimes L_2 \\ = & \{w \in (\Sigma_1 \cup \Sigma_2)^\infty \mid w|_{\Sigma_1} \in L_1 \text{ and } w|_{\Sigma_2} \in L_2\} \\ = & \{w \in (\Sigma_2 \cup \Sigma_1)^\infty \mid w|_{\Sigma_2} \in L_2 \text{ and } w|_{\Sigma_1} \in L_1\} \\ = & L_2 \otimes L_1 \end{aligned}$$

□

7.3 Channel-Guard Languages

The language of a LOOP+ ω program as well as its natural over-approximation describes its behavior from a local, reactive point of view. It is basically a description of how the program reacts (in terms of I/O-operations) to the reading of a certain symbol from its input channels. The language does, however, *not* describe the rules which apply to channel systems. In particular, it does not describe that a symbol a can only be read from a channel $C_{J \rightarrow I}$ if the symbol was sent to that channel beforehand. This is due to the fact that the language of the program of program

P_I contains only the receive symbols $|J \xrightarrow{a} I\rangle$ and $|J \xrightarrow{b} I\rangle$, but not the according send symbols $\langle J \xrightarrow{a} I|$ and $\langle J \xrightarrow{b} I|$ which are associated to the channel $C_{J \rightarrow I}$.

The consequence of this is that the language $\text{Interleaving}(\{P_1, \dots, P_N\})$, as described in the section before, contains words which resemble a behavior which does not obey the rules of channel systems. We do, however, have to respect these rules, which dictate three things:

1. At any point in time, a program P_I can read only exactly those symbols from channel $C_{J \rightarrow I}$ which the program P_J has send to that channel beforehand.
2. The channel system respects the FIFO principle: The symbols can only be read from a channel in exactly the same order as they were sent to that channel.
3. Only if all symbols which were sent to the channel $C_{J \rightarrow I}$ are read from that channel, the channel is empty and a reading attempt generates the symbol $|J \xrightarrow{\perp} I\rangle$.

The difficulty of guarding the rules above lies deep and is due to the fact that the channels are generally unbounded: Such unbounded channels can be used to simulate tapes of Turing Machines. Even if the programs behavior itself is limited to some regular behavior (e.g. by the natural over–approximation), the problem due to the unbounded channels remains. The reason for that is that DFA which are extended by the ability to communicate over unbounded channels (often referred to as *communicating automata* or *communicating finite state machines*) are able to simulate the behavior of any Turing Machine [BZ83], which of course renders reachability questions and alike undecidable. Also, deciding whether the channel capacity is inherently bounded in a specific setup is undecidable, as this would be equivalent deciding boundedness for Turing Machines, which as is know is undecidable.

The conclusion we draw from that is that we will simply bound the channels, so to say *by force*, in order to make the network of communicating programs analyzable. This has the convenient side effect that we obtain an ω –regular channel guard language. Such ω –regular channel guard languages can be described by means of a special Büchi Automaton as follows:

Definition D.7.5 (CHANNEL GUARD LANGUAGES):

The **channel guard automaton** $\mathfrak{A}_{C_{J \rightarrow I}}^k$ is a non–deterministic Büchi Automaton $\mathfrak{A}_{C_{J \rightarrow I}}^k = (Q, \Sigma, \Delta, q_0, F)$ where

$$Q = \left(\bigcup_{i \in \{0, \dots, k\}} \{a, b\}^i \right) \cup \left(\tilde{a} \cdot \{a, b\}^k \cup \tilde{b} \cdot \{a, b\}^k \right)$$

$$\Sigma = \left\{ \langle J \xrightarrow{x} I|, \langle \widetilde{J \xrightarrow{x} I}|, |J \xrightarrow{x} I\rangle, |J \xrightarrow{\perp} I\rangle, \overset{x}{\sim} \right. \\ \left. \mid x \in \{a, b\}, X \in \{I, J\} \right\}$$

$$q_0 = \varepsilon,$$

$$F = Q,$$

and Δ is given as follows:

$$\left(w, \wr J \xrightarrow{x} I |, x \cdot w \right) \in \Delta, \quad \text{if } |w| < k \quad (\text{S.I})$$

$$\left(w, \wr J \xrightarrow{x} I |, \tilde{x} \cdot w \right) \in \Delta, \quad \text{if } |w| = k \quad (\text{S.II})$$

$$\left(\tilde{x} \cdot w, \wr \widetilde{J \xrightarrow{x} I} |, w \right) \in \Delta \quad (\text{S.III})$$

$$\left(w \cdot x, | J \xrightarrow{x} I \circlearrowleft, w \right) \in \Delta \quad (\text{R.I})$$

$$\left(\varepsilon, | J \xrightarrow{\perp} I \circlearrowleft, \varepsilon \right) \in \Delta \quad (\text{R.II})$$

$$\left(w, \widetilde{\sim}^X \widetilde{\sim}, w \right) \in \Delta, \quad \text{for } X \in \{I, J\} \quad (\text{P})$$

We call the language $L(\mathfrak{A}_{C_{J \rightarrow I}}^k)$ which is generated by the automaton $\mathfrak{A}_{C_{J \rightarrow I}}^k$ the **channel guard language of channel $C_{J \rightarrow I}$ with channel size k** and denote this language by $L_{C_{J \rightarrow I}}^k$.

We now explain Definition D.7.5 in more detail: The state space of $\mathfrak{A}_{C_{J \rightarrow I}}^k$ consists of all words of a length of at most k over the alphabet $\{a, b\}$ as well as words of the form $\tilde{x} \cdot \{a, b\}^k$ where $x \in \{a, b\}$. The automaton $\mathfrak{A}_{C_{J \rightarrow I}}^k$ uses this state space to emulate a channel whose size is bounded by k . The words of the form $\tilde{x} \cdot \{a, b\}^k$ are used, to memorize a symbol x if attempting to put x in a full channel.

Obviously the automaton $\mathfrak{A}_{C_{J \rightarrow I}}^k$ starts with an empty channel as the initial state. Furthermore, since the state space contains only legal configurations of the channel, every state is a final state. That does however not mean that $\mathfrak{A}_{C_{J \rightarrow I}}^k$ accepts every word, as we leave out the illegal moves in the transition relation.

The alphabet of $\mathfrak{A}_{C_{J \rightarrow I}}^k$ consists of all symbols from $\Sigma_{I/O}$ which indicate some event on channel $C_{J \rightarrow I}$, namely P_J sending a symbol to P_I ($\wr J \xrightarrow{a} I |$ and $\wr J \xrightarrow{b} I |$), P_I reading a symbol from channel $C_{J \rightarrow I}$ ($| J \xrightarrow{a} I \circlearrowleft$ and $| J \xrightarrow{b} I \circlearrowleft$) and P_I attempting to read a symbol from the empty channel $C_{J \rightarrow I}$ ($| J \xrightarrow{\perp} I \circlearrowleft$). Furthermore, the alphabet of $\mathfrak{A}_{C_{J \rightarrow I}}^k$ contains two symbols which indicate that P_J sends a symbol to P_I but the channel $C_{J \rightarrow I}$ is already full ($\wr \widetilde{J \xrightarrow{a} I} |$ and $\wr \widetilde{J \xrightarrow{b} I} |$). Additionally, to represent idleness on the channel, we need the pulse symbols of program P_I and P_J ($\widetilde{\sim}^I$ and $\widetilde{\sim}^J$).

Finally, the transition relation Δ encodes the logic of the automaton $\mathfrak{A}_{C_{J \rightarrow I}}^k$. In the following we explain every of the given rules for Δ :

(S.I) If P_J sends a symbol x to the *non-full* channel $C_{J \rightarrow I}$ whose content is currently w , then the channel content is updated to $x \cdot w$ and thus the new state is $x \cdot w$. The symbol $\wr J \xrightarrow{x} I |$ indicates the sending event.

(S.II) If P_J sends a symbol x to the *full* channel $C_{J \rightarrow I}$ whose content is currently w , then the channel content is left as is and the symbol x is memorized, and thus the new state is $\tilde{x} \cdot w$. The symbol $\wr J \xrightarrow{x} I |$ again indicates the sending event.

- (S.III) If the automaton recently witnessed an attempt to send a symbol x to a full channel with content w and has therefore currently memorized the symbol x (i.e. the automaton is currently in state $\tilde{x} \cdot w$), then the next symbol *must* be the symbol $\langle J \xrightarrow{x} I \mid$ which indicates this failed attempt. This way, though the symbol x which was sent from P_J to P_I is effectively lost for P_I , this incident is at least indicated. After indication, the new state is w .
- (R.I) If P_I reads a symbol from the channel $C_{J \rightarrow I}$ whose content is currently $w \cdot x$ then the next symbol *must* be the symbol $\mid J \xrightarrow{x} I \rangle$ which indicates the reading event.
- (R.II) If P_I reads a symbol from the *empty* channel $C_{J \rightarrow I}$, i.e. the channels content is currently ε , then the next symbol *must* be the symbol $\mid J \xrightarrow{\perp} x \rangle I$ which indicates the failed reading attempt.
- (P) The program X emitting its pulse symbol $\sim \overset{x}{\sim} \sim$ has no effect on the channel, thus the current state is simply revisited.

To gain a little more insight in how such a channel guard automaton works, we take a look at the following example:

Example E.7.1 (CHANNEL GUARD AUTOMATA):

Consider the channel guard automaton $\mathfrak{A}_{C_{J \rightarrow I}}^1$. Furthermore consider a program P_J which sends the symbols a , b and a (in that order) to program P_I (i.e. to channel $C_{J \rightarrow I}$) and then idles. The language of such a program P_J is given by

$$L(P_J) = \{ \langle J \xrightarrow{a} I \mid \cdot \langle J \xrightarrow{b} I \mid \cdot \langle J \xrightarrow{a} I \mid \cdot (\sim \overset{J}{\sim})^\omega \}$$

The program P_I is a program, that keeps reading from channel $C_{J \rightarrow I}$ until it reads the symbol b . Then it idles. The language of such a program P_I is given by

$$L(P_I) = \left(\left\{ \mid J \xrightarrow{\perp} I \rangle + \mid J \xrightarrow{a} I \rangle \right\} \cdot \{ \sim \overset{I}{\sim} \} \right)^* \cdot \left\{ \mid J \xrightarrow{b} I \rangle \right\} \cdot \{ \sim \overset{J}{\sim} \}^\omega \\ \cup \left(\left\{ \mid J \xrightarrow{\perp} I \rangle + \mid J \xrightarrow{a} I \rangle \right\} \cdot \{ \sim \overset{I}{\sim} \} \right)^\omega$$

We can now synchronize the languages $L_{C_{J \rightarrow I}}^1 = L(\mathfrak{A}_{C_{J \rightarrow I}}^1)$, $L(P_I)$ and $L(P_J)$ to obtain $L = L_{C_{J \rightarrow I}}^1 \otimes L(P_I) \otimes L(P_J)$. The language $L_{C_{J \rightarrow I}}^1$ guards the channel $C_{J \rightarrow I}$ and bounds the channel's size to 1. In other words, the synchronization with $L_{C_{J \rightarrow I}}^1$ ensures that L respects the rules of channel systems.

For instance, no word in L starts with the symbol $\mid J \xrightarrow{a} I \rangle$ or $\mid J \xrightarrow{b} I \rangle$, even though there are words in $L(P_I)$ which do start with the symbol $\mid J \xrightarrow{a} I \rangle$ or $\mid J \xrightarrow{b} I \rangle$. The synchronization with $L_{C_{J \rightarrow I}}^1$ prevents that a symbol is read from the channel before it was sent to that channel.

Another example is that no word in L starts with $\langle J \xrightarrow{a} I \mid \cdot \langle J \xrightarrow{b} I \mid \cdot \mid J \xrightarrow{a} I \rangle \mid J \xrightarrow{b} I \rangle$. This is because in such words the channel size

of 1 is not respected. Instead, all words in L that do start with $\langle J \xrightarrow{a} I \mid \cdot \langle J \xrightarrow{b} I \mid$ inevitably have $\langle J \xrightarrow{b} I \mid$ as the next symbol, indicating that P_J tried to send the symbol b to the full channel $C_{J \rightarrow I}$ of program P_I .

As we have learned, channel guard languages ensure that the logic of a channel system is respected. We can define a language which guards every channel in a network of N communicating LOOP+ ω programs P_1, \dots, P_N as

$$\mathbf{Guard}^k(\{P_1, \dots, P_N\}) = \bigotimes_{I, J \in \{1, \dots, N\}} \left(L_{C_{J \rightarrow I}}^k \right).$$

In a next step we can then refine the aether language — i.e. the communication behavior of the network of N communicating LOOP+ ω programs P_1, \dots, P_N — to

$$\mathbf{Aether}^k(\{P_1, \dots, P_N\}) = \mathbf{Interleaving}(\{P_1, \dots, P_N\}) \otimes \mathbf{Guard}^k(\{P_1, \dots, P_N\}),$$

for some constant channel size of our choice k . It would of course be desirable that the intersection

$$\mathbf{Aether}^k(\{P_1, \dots, P_N\}) \cap \left(\Sigma_{I/O}^* \cdot \left\{ \langle I \xrightarrow{x} J \mid \mid x \in \{a, b\} \text{ and } I, J \in \{1, \dots, N\} \right\} \cdot \Sigma_{I/O}^\omega \right)$$

is empty, because then the channel size k was chosen great enough so that no channel overflows occur. Unfortunately, as mentioned before, there is no way of deciding whether such a bound k even exists such that this intersection becomes empty. It is, however, possible to successively increase the channel size k and test the above intersection for emptiness after each increase. In case of emptiness, a sufficiently great channel size is found. Otherwise, k has to be increased further. This procedure does, however, obviously not terminate necessarily as the channel size might not be bounded. In case of Example E.7.1 however, this method would have successfully established the channel bound 3.

7.4 Fairness–Ensuring Languages

So far, we have already described how to correctly over–approximate the communication behavior of a network of communicating programs which communicate over channels.

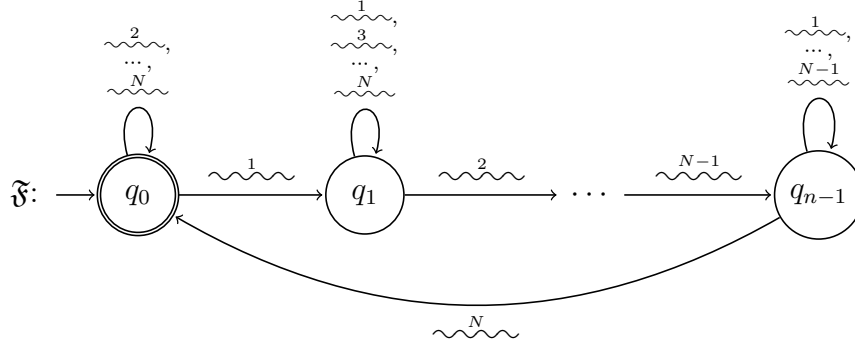
Up until now, however, our model lacks a notion of fairness, for there are words in our model which represent scenarios in which a program may never proceed with its calculations. We can exclude such unfair behavior from $\mathbf{Aether}^k(\{P_1, \dots, P_N\})$ by intersecting or rather synchronizing the language $\mathbf{Aether}^k(\{P_1, \dots, P_N\})$ with some fairness–ensuring language $\mathbf{Fair}(\{P_1, \dots, P_N\})$.

The language $\mathbf{Fair}(\{P_1, \dots, P_N\})$ ensures that each of the programs P_1, \dots, P_N eventually proceeds with its computation. This is fairly easy to ensure as we have

the obligatory pulse in the language of each program. If the program proceeds with its computation, it will eventually generate a pulse symbol. The property that all programs will eventually generate a pulse signal is ω -regular, as we can provide a Büchi Automaton which ensures this:

Definition D.7.6 (FAIRNESS-ENSURING LANGUAGE):

The fairness-ensuring language $\mathfrak{Fair}(\{P_1, \dots, P_N\})$ for a network of N communicating LOOP+ ω programs P_1, \dots, P_N is given by the language $L(\mathfrak{F})$ of the following Büchi Automaton \mathfrak{F} :



If we now have an aether language $\mathfrak{Aether}^k(\{P_1, \dots, P_N\})$ at hand and want to ensure fairness amongst the programs, we simply calculate the synchronization of $\mathfrak{Aether}^k(\{P_1, \dots, P_N\})$ and $\mathfrak{Fair}(\{P_1, \dots, P_N\})$ and obtain

$$\mathfrak{Aether}_{\text{fair}}^k(\{P_1, \dots, P_N\}) = \mathfrak{Aether}^k(\{P_1, \dots, P_N\}) \otimes \mathfrak{Fair}(\{P_1, \dots, P_N\})$$

The drawback is, however, that there is no bound on how much slower one program may proceed with its computation than the other programs. This can result in the unrealistic scenario that one program does its calculations arbitrarily faster than some other program. A fairly easy way to circumvent this problem is to modify the LOOP+ ω semantics so that the execution of *every* statement generates a pulse symbol.

7.5 Application: Checking for Deadlocks

Finally, we want to give an example of how a network of communicating LOOP+ ω programs can be checked for unwanted behavior. For instance, we might want to assert that a system of communicating LOOP+ ω programs is deadlock-free. The occurrence of a deadlock in a network of N communicating programs P_1, \dots, P_N can be formalized by the following ω -regular language:

$$\mathfrak{Deadlock}(\{P_1, \dots, P_N\}) = \Sigma_{I/O}^* \cdot \left\{ I \xrightarrow{\perp} J, \sim^I \mid I, J \in \{1, \dots, N\} \right\}^\omega$$

In every word in $\mathfrak{Deadlock}(\{P_1, \dots, P_N\})$ from some point onwards only failed read-attempts (and obligatory pulse symbols) occur. This formalizes precisely a global deadlock situation in which every program waits for input from another program before it proceeds with its calculation. If we now want to assert that no deadlock

occurs in a network of N communicating programs P_1, \dots, P_N , we simply check the intersection

$$\mathfrak{Aether}_{\text{fair}}^k(\{P_1, \dots, P_N\}) \cap \mathfrak{Deadlock}(\{P_1, \dots, P_N\})$$

for emptiness. If this intersection is empty, then in fact no deadlock occurs in the network. If the intersection is non-empty, then the intersection contains possible candidates for deadlock situations. Note that the network can still behave deadlock-free even if the intersection is non-empty, since $\mathfrak{Aether}_{\text{fair}}^k(\{P_1, \dots, P_N\})$ is an *over-approximation* of the actual behavior of the network. If the intersection is empty however, then the network's behavior is actually deadlock-free.

8

Conclusion

8.1 Outlook

As we have seen in Example E.6.1, the natural over-approximation might become quite coarse even for very simple programs. In the following we therefore want to mention a few ideas and suggestions on how to improve on the over-approximation of a LOOP+ ω program's communication behavior:

A very simple idea is to violently restrict the domain of the program variables to a finite domain. It is quite easy to see that this would result in a ω -regular behavior of the program.

We can also establish that the communication behavior of a LOOP+ ω program without inner loops inside the ω -loop is ω -regular, however, both a finite variable domain as well as loop-loop-free LOOP+ ω programs are not realistic assumptions.

A more constructive approach would be to perform a static analysis like constant propagation or interval analysis to potentially establish that some variables in a program de facto only evaluate to a finite range of values. This could refine the over-approximation especially for loop-loops which are otherwise simply over-approximated by any *arbitrary* (but finite) number of loop iterations.

Another meaningful analysis might be to analyze whether the set of valuations which are reachable by executing the ω -loop body is finite given an arbitrary starting valuation. If so, then the program's behavior is also ω -regular. This problem is equivalent to determining whether the image of a primitive recursive function is finite, which is, however, undecidable as a consequence of the Kleene Normal Form Theorem. We could, however, approach this problem by a symbolic execution of the LOOP+ ω program. If a symbolic execution establishes that only a finite set of valuations of the program variables is reachable, then the program's behavior is in fact ω -regular.

8.2 Summary

In this master thesis we have introduced a novel model programming language for communicating channel systems called LOOP+ ω , whose programs consist of exactly one outer infinite loop and any composition and nesting of primitive recursive, thus bounded, inner loops. We have described the syntax of LOOP+ ω by extending the well-known model program language LOOP by message passing capabilities and infinite loops. Also, we have formalized denotational semantics of LOOP+ ω programs in terms of an I/O-language which describes a program's communication behavior.

We have then studied the expressiveness of LOOP+ ω as well as a few decision problems for the class of LOOP+ ω generated languages hoping that some decision problems like e.g. the Inclusion Problem or the Equivalence Problem might become decidable through the restriction of *guaranteed non-termination* of the outer loop and *guaranteed termination* of all inner loops. We have shown that our hopes were dashed by the Kleene Normal Form Theorem which states that every WHILE program can be rewritten using only one outer unbounded while-loop and only primitive recursive inner loop-loops. Furthermore we were able to adapt Rice's Theorem to LOOP+ ω programs rendering most decision problems for LOOP+ ω programs undecidable.

In order to still be able to analyze the communication behavior of a program, we have provided a method of obtaining a correct ω -regular over-approximation of a LOOP+ ω program's communication behavior. We have then shown how to interleave and synchronize the languages of N communicating LOOP+ ω programs while obeying the logics of channel systems and ensuring fairness amongst the programs in order to obtain an ω -regular model of a network of communicating LOOP+ ω programs. We have also shown how to analyze such a model of the network's communication behavior in the context of deadlock detection.

Finally we have listed a few ideas and suggestions on how to refine our analysis against the background of the potential coarseness of an ω -regular over-approximation of a LOOP+ ω program's communication behavior.

A

Appendix — Omitted Proofs

A.1 Proof of Lemma L.4.2

Lemma L.4.2 (PREFIX PRESERVATION BY LOOP AND LOOP+ ω SEMANTICS):

For any LOOP or LOOP+ ω program $P \in \mathbb{L} \cup \mathbb{L}_\omega$, any set of configurations created by ω -loop-free programs $S \subseteq \mathbb{S}$ and any word $w \in \Sigma_{I/O}^$ it holds that*

$$\llbracket P \rrbracket(w \cdot S) = w \cdot \llbracket P \rrbracket(S).$$

Proof: Again, the proof of this lemma is done by structural induction over all LOOP and LOOP+ ω programs, very much like the proof of Lemma L.4.1. For the induction basis, we have the non-composed statements. As the induction hypothesis we assume that Lemma L.4.2 holds for programs P_1 , P_2 and P_3 . We then show that, given the induction hypothesis, Lemma L.4.2 also holds for the statements composed of P_1 , P_2 and P_3 .

Induction Basis

For the induction basis we prove Lemma L.4.2 for every non-composed statement:

Empty Statement

$$\llbracket \text{skip} \rrbracket(w \cdot S) = w \cdot S \tag{D.3.8}$$

$$= w \cdot \llbracket \text{skip} \rrbracket(S) \tag{D.3.8}$$

Assignment

$$\llbracket x_i := x_j + c \rrbracket(w \cdot S) = \llbracket x_i := x_j + c \rrbracket(\{(\sigma, w \cdot v) \mid (\sigma, v) \in S\}) \tag{D.3.5}$$

$$= \{(\sigma[x_i \leftarrow \sigma(x_j) + c], w \cdot v) \mid (\sigma, v) \in S\} \quad (\text{D.3.8})$$

$$= w \cdot \{(\sigma[x_i \leftarrow \sigma(x_j) + c], v) \mid (\sigma, v) \in S\} \quad (\text{D.3.5})$$

$$= w \cdot \llbracket x_i := x_j + c \rrbracket(S) \quad (\text{D.3.8})$$

Message Sending and Pulsing

$$\llbracket \text{send}_J \text{ a} \rrbracket(w \cdot S) = (w \cdot S) \cdot \wr I \xrightarrow{a} J \mid \quad (\text{D.3.8})$$

$$= w \cdot (S \cdot \wr I \xrightarrow{a} J \mid) \quad (\text{C.3.1})$$

$$= w \cdot \llbracket \text{send}_J \text{ a} \rrbracket(S) \quad (\text{D.3.8})$$

The case for `sendJ b` and `pulse` is completely analogous.

We now have shown that for all non-composed statements Lemma L.4.2 holds which concludes the proof for the induction basis.

Induction Hypothesis

As our induction hypothesis we assume that Lemma L.4.2 holds for the LOOP programs P_1 , P_2 and P_3 . More formally we assume $\llbracket P_1 \rrbracket(w \cdot S) = w \cdot \llbracket P_1 \rrbracket(S)$, $\llbracket P_2 \rrbracket(w \cdot S) = w \cdot \llbracket P_2 \rrbracket(S)$ and $\llbracket P_3 \rrbracket(w \cdot S) = w \cdot \llbracket P_3 \rrbracket(S)$.

Induction Step

We now prove that, given the induction hypothesis, Lemma L.4.2 holds for every composed statement.

Message Reception

$$\begin{aligned} & \llbracket \text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END} \rrbracket(w \cdot S) \\ &= \llbracket P_1 \rrbracket \left((w \cdot S) \cdot |J \xrightarrow{a} I \wr \right) \cup \llbracket P_2 \rrbracket \left((w \cdot S) \cdot |J \xrightarrow{b} I \wr \right) \quad (\text{D.3.8}) \\ & \quad \cup \llbracket P_3 \rrbracket \left((w \cdot S) \cdot |J \xrightarrow{\perp} I \wr \right) \end{aligned}$$

$$\begin{aligned} &= \llbracket P_1 \rrbracket \left(w \cdot (S \cdot |J \xrightarrow{a} I \wr) \right) \cup \llbracket P_2 \rrbracket \left(w \cdot (S \cdot |J \xrightarrow{b} I \wr) \right) \quad (\text{C.3.1}) \\ & \quad \cup \llbracket P_3 \rrbracket \left(w \cdot (S \cdot |J \xrightarrow{\perp} I \wr) \right) \end{aligned}$$

$$\begin{aligned} &= w \cdot \llbracket P_1 \rrbracket \left(S \cdot |J \xrightarrow{a} I \wr \right) \cup w \cdot \llbracket P_2 \rrbracket \left(S \cdot |J \xrightarrow{b} I \wr \right) \quad (\text{I.H.}) \\ & \quad \cup w \cdot \llbracket P_3 \rrbracket \left(S \cdot |J \xrightarrow{\perp} I \wr \right) \end{aligned}$$

$$\begin{aligned} &= w \cdot \left(\llbracket P_1 \rrbracket \left(S \cdot |J \xrightarrow{a} I \wr \right) \cup \llbracket P_2 \rrbracket \left(S \cdot |J \xrightarrow{b} I \wr \right) \right) \quad (\text{C.4.3}) \\ & \quad \cup \llbracket P_3 \rrbracket \left(S \cdot |J \xrightarrow{\perp} I \wr \right) \end{aligned}$$

$$= w \cdot \llbracket \text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END} \rrbracket(S) \quad (\text{D.3.8})$$

Concatenation

$$\begin{aligned}
\llbracket P_1; P_2 \rrbracket(w \cdot S) &= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(w \cdot S)) && \text{(D.3.8)} \\
&= \llbracket P_2 \rrbracket(w \cdot \llbracket P_1 \rrbracket(S)) && \text{(I.H.)} \\
&= w \cdot \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(S)) && \text{(I.H.)} \\
&= w \cdot \llbracket P_1; P_2 \rrbracket(S) && \text{(D.3.8)}
\end{aligned}$$

Loops

$$\begin{aligned}
&\llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket(w \cdot S) \\
&= \llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket(\{(\sigma, w \cdot v) \mid (\sigma, v) \in S\}) && \text{(D.3.5)} \\
&= \bigcup_{(\sigma, w \cdot v) \in w \cdot S} \left(\llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, w \cdot v)\}) \right) && \text{(D.3.8, D.3.5)} \\
&= \bigcup_{(\sigma, w \cdot v) \in w \cdot S} \left(\llbracket P_1 \rrbracket^{\sigma(x)}(w \cdot \{(\sigma, v)\}) \right) && \text{(D.3.5)}
\end{aligned}$$

The expression $\llbracket P_1 \rrbracket^{\sigma(x)}(w \cdot S)$ represents a $\sigma(x)$ -fold application of P_1 to $w \cdot S$ which is the same as $\underbrace{\llbracket P_1; \dots; P_1 \rrbracket}_{\sigma(x) \text{ times}}(w \cdot S)$, so we can use the case for the concatenation:

$$\begin{aligned}
&= \bigcup_{(\sigma, w \cdot v) \in w \cdot S} \left(w \cdot \llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, v)\}) \right) \\
&= w \cdot \left(\bigcup_{(\sigma, w \cdot v) \in w \cdot S} \left(\llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, v)\}) \right) \right) && \text{(C.4.3)} \\
&= w \cdot \left(\bigcup_{(\sigma, v) \in S} \left(\llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, v)\}) \right) \right) && \text{(D.3.5)} \\
&= w \cdot \llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket(S) && \text{(D.3.8)}
\end{aligned}$$

 ω -Loops

$$\begin{aligned}
&\llbracket \text{LOOP } \omega \text{ DO } P_1 \text{ END} \rrbracket(w \cdot S) \\
&= \left\{ (\perp, v) \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \llbracket P_1 \rrbracket^k(w \cdot S) \end{array} \right. \right\} && \text{(D.3.8)}
\end{aligned}$$

The expression $\llbracket P_1 \rrbracket^k(w \cdot S)$ represents a k -fold application of P_1 to $w \cdot S$ which is the same as $\underbrace{\llbracket P_1; \dots; P_1 \rrbracket}_{k \text{ times}}(w \cdot S)$, so we can use the case for the concatenation:

$$= \left\{ (\perp, v) \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in w \cdot \llbracket P_1 \rrbracket^k(S) \end{array} \right. \right\}$$

Every word $v_0 \cdot v_1 \cdots v_k$ must have the prefix w . Also there exists a smallest number $i \in \mathbb{N}$ such that w is still a prefix of $v_0 \cdots v_i$. We can thus rewrite v as $w \cdot v'_i \cdot v_{i+1} \cdots$ where v'_i is a suffix of v_i . We then do an index shift on the i 's and thereby obtain the ω -word $z = z_0 \cdot z_1 \cdots$ with $v'_i = z_0$, $v_{i+1} = z_1$, $v_{i+2} = z_2$, \dots and with $v = w \cdot z$:

$$\begin{aligned}
&= \left\{ (\perp, w \cdot z) \left| \begin{array}{l} w \cdot z \in \Sigma_{I/O}^\omega, \exists z_0 \cdot z_1 \cdot z_2 \cdots = z, \\ \text{such that } z_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w \cdot z_0 \cdot z_1 \cdots z_k) \in w \cdot \llbracket P_1 \rrbracket^k(S) \end{array} \right. \right\} \\
&= w \cdot \left\{ (\perp, z) \left| \begin{array}{l} w \cdot z \in \Sigma_{I/O}^\omega, \exists z_0 \cdot z_1 \cdot z_2 \cdots = z, \\ \text{such that } z_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, w \cdot z_0 \cdot z_1 \cdots z_k) \in w \cdot \llbracket P_1 \rrbracket^k(S) \end{array} \right. \right\} \quad (\text{D.3.5})
\end{aligned}$$

In this expression, we can drop w inside the curly braces, basically by a well-thought-out application of Definition D.3.5:

$$\begin{aligned}
&= w \cdot \left\{ (\perp, z) \left| \begin{array}{l} z \in \Sigma_{I/O}^\omega, \exists z_0 \cdot z_1 \cdot z_2 \cdots = z, \\ \text{such that } z_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, z_0 \cdot z_1 \cdots z_k) \in \llbracket P_1 \rrbracket^k(S) \end{array} \right. \right\} \\
&= w \cdot \llbracket \text{LOOP } \omega \text{ DO } P_1 \text{ END} \rrbracket(S) \quad (\text{D.3.8})
\end{aligned}$$

This concludes the proof for ω -loops and thereby concludes the proof of Lemma L.4.2. \square

A.2 Proof of Lemma L.4.5

Lemma L.4.5:

Let P be a LOOP or LOOP+ ω program and let $S_\varepsilon = \{(\sigma, \varepsilon)\}$ be a configuration containing only one valuation-word-pair where the word is the empty word. Then $L_{S_\varepsilon}(P)$ is reactive.

Proof: We prove reactivity of $L_{S_\varepsilon}(P)$ by structural induction over all LOOP and all LOOP+ ω programs. For the induction basis, we have the non-composed statements. As the induction hypothesis we assume that Lemma L.4.5 holds for programs P_1 , P_2 and P_3 . We then show that, given the induction hypothesis, Lemma L.4.5 also holds for the statements composed of P_1 , P_2 and P_3 .

Induction Basis

For the induction basis, we prove Lemma L.4.5 holds for every non-composed statement: The language of a program that consists only of a single non-composed statement is either the empty word or a single symbol. In both cases the language is trivially reactive.

Induction Hypothesis

As our induction hypothesis we assume that Lemma L.4.5 holds for the LOOP programs P_1 , P_2 and P_3 . More formally we assume that $L(P_1)$, $L(P_2)$ and $L(P_3)$ are all

reactive.

Induction Step

For the induction step it remains to show that, given the induction hypothesis, Lemma L.4.5 holds for the composed statements.

Message Reception

$$\begin{aligned} & L_{S_\varepsilon}(\text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END}) \\ &= L_{S_\varepsilon \cdot |J \xrightarrow{a} I \rangle}(P_1) + L_{S_\varepsilon \cdot |J \xrightarrow{b} I \rangle}(P_2) + L_{S_\varepsilon \cdot |J \xrightarrow{\perp} I \rangle}(P_3) \end{aligned} \quad (\text{L.4.3})$$

$$= L_{|J \xrightarrow{a} I \rangle \cdot S_\varepsilon}(P_1) + L_{|J \xrightarrow{b} I \rangle \cdot S_\varepsilon}(P_2) + L_{|J \xrightarrow{\perp} I \rangle \cdot S_\varepsilon}(P_3) \quad (\text{C.4.6})$$

$$= |J \xrightarrow{a} I \rangle \cdot L_{S_\varepsilon}(P_1) + |J \xrightarrow{b} I \rangle \cdot L_{S_\varepsilon}(P_2) + |J \xrightarrow{\perp} I \rangle \cdot L_{S_\varepsilon}(P_3) \quad (\text{C.4.5})$$

The languages $L_{S_\varepsilon}(P_1)$, $L_{S_\varepsilon}(P_2)$ and $L_{S_\varepsilon}(P_3)$ are reactive by means of the induction hypothesis. The symbols appended to their front are input characters, so we have to check whether the first requirement of Definition D.4.3 is satisfied: Indeed, the requirement is satisfied because for every single input character $|J \xrightarrow{a} I \rangle$, $|J \xrightarrow{b} I \rangle$, and $|J \xrightarrow{\perp} I \rangle$, there is at least one word in $L(\text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END})$ starting with that input character.

Concatenation

$$\begin{aligned} & L_{S_\varepsilon}(P_1 ; P_2) \\ &= \bigoplus_{(\tau, v) \in \llbracket P_1 \rrbracket(S_\varepsilon)} \left(v \cdot L_{\{(\tau, \varepsilon)\}}(P_2) \right) \end{aligned} \quad (\text{L.4.3})$$

At this point, we need a short digression:

Digression

In order to proceed with the proof, we first need to prove the following:

$$\forall(\sigma, w), (\tau, v) \in \llbracket P \rrbracket(S_\varepsilon): ((w = v) \implies (\sigma = \tau))$$

We prove this again by structural induction over all LOOP+ ω programs: For the induction basis we have the non-composed statements: These statements transform the configuration $S_\varepsilon = \{(\sigma, \varepsilon)\}$ to $\{\sigma', \varepsilon\}$ (empty statement, assignment) or to $\{\sigma, a\}$ (message sending and pulsing). Both of these resulting configurations are singleton sets for which the claim trivially holds.

As the induction hypothesis, we assume that the above claim holds for the programs P_1 , P_2 and P_3 . For the induction step, we have to show that, given the induction hypothesis, the above claim holds for the composed statements as well:

Digression (cont)

First, we examine message reception. The message reception statement $\text{RECV}_J \text{ a DO } P_1 \text{ b DO } P_2 \text{ _ DO } P_3 \text{ END}$ transforms the configuration S_ε into

$$\left\{ |J \xrightarrow{a} I \rangle \cdot \llbracket P_1 \rrbracket(\sigma, \varepsilon), |J \xrightarrow{b} I \rangle \cdot \llbracket P_2 \rrbracket(\sigma, \varepsilon), |J \xrightarrow{\perp} I \rangle \cdot \llbracket P_3 \rrbracket(\sigma, \varepsilon) \right\}.$$

In this configuration, all words are different, so the claim trivially holds.

Next, we examine the concatenation. The concatenation of two programs $P_1; P_2$ transforms the configuration S_ε into

$$\llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(\sigma, \varepsilon)).$$

Now by the induction hypothesis, the claim holds for the configuration $S' = \llbracket P_1 \rrbracket(\sigma, \varepsilon)$. As the claim holds, we can partition S' into $S'_1 \cup \dots \cup S'_k$ such that each partition S'_i is associated to only one word and no two partitions are associated to the same word. We then execute $\llbracket P_2 \rrbracket$ on each partition and, by induction hypothesis, for each $\llbracket P_2 \rrbracket(S'_i)$ obtain a result which satisfies the claim as well. As semantics functions preserve prefixes (Lemma L.4.2), the configurations $\llbracket P_2 \rrbracket(S'_i)$ all have different prefixes and therefore the union $\llbracket P_2 \rrbracket(S'_1) \cup \dots \cup \llbracket P_2 \rrbracket(S'_k)$ satisfies the claim as well. It holds that

$$\begin{aligned} & \llbracket P_2 \rrbracket(S'_1) \cup \dots \cup \llbracket P_2 \rrbracket(S'_k) \\ &= \llbracket P_2 \rrbracket(S') \\ &= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(S_\varepsilon)) \\ &= \llbracket P_1; P_2 \rrbracket(S_\varepsilon) \end{aligned} \tag{L.4.1}$$

and therefore the claim holds for the concatenation as well.

Next, we examine loop-loops. The loop $\text{LOOP } x \text{ DO } P_1 \text{ END}$ transforms the configuration S_ε into

$$= \bigcup_{(\sigma, w) \in S_\varepsilon} \llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, w)\}),$$

which simplifies as follows:

$$\begin{aligned} &= \bigcup_{(\sigma, w) \in \{(\sigma, \varepsilon)\}} \llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, w)\}) \\ &= \llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, \varepsilon)\}) \\ &= \underbrace{\llbracket P_1 \rrbracket; \dots; \llbracket P_1 \rrbracket}_{\sigma(x) \text{ times}}(\{(\sigma, \varepsilon)\}) \end{aligned}$$

which resembles the case for concatenation.

Finally, we examine the ω -loop: According to Definition D.3.8 the ω -loop statement transforms every configuration into a configuration in which all configurations are \perp . Therefore, the claim trivially holds for the ω -loop as well.

According to the digression above, the valuation function τ is only dependent on the word v and therefore all v in the union above are different. If all v are different, then the reactiveness of $L(P_1; P_2)$ depends only on the reactiveness of $L_{\{(\tau, \varepsilon)\}}(P_2)$ which is given by the induction hypothesis.

Loops

$$\begin{aligned} & L_{S_\varepsilon}(\text{LOOP } x \text{ DO } P_1 \text{ END}) \\ &= \{w \mid (\tau, w) \in \llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket(S_\varepsilon)\} \end{aligned} \quad (\text{D.3.10})$$

$$\begin{aligned} &= \{w \mid (\tau, w) \in \llbracket \text{LOOP } x \text{ DO } P_1 \text{ END} \rrbracket(\{(\sigma, \varepsilon)\})\} \\ &= \left\{ w \mid (\tau, w) \in \bigcup_{(\nu, \varepsilon) \in \{(\sigma, \varepsilon)\}} \llbracket P_1 \rrbracket^{\nu(x)}(\{(\nu, \varepsilon)\}) \right\} \quad (\text{D.3.8}) \\ &= \{w \mid (\tau, w) \in \llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, \varepsilon)\})\} \end{aligned}$$

The expression $\llbracket P_1 \rrbracket^{\sigma(x)}(\{(\sigma, \varepsilon)\})$ represents a $\sigma(x)$ -fold application of P_1 to $\{(\sigma, \varepsilon)\}$ which is the same as $\underbrace{\llbracket P_1; \dots; P_1 \rrbracket}_{\sigma(x) \text{ times}}(\{(\sigma, \varepsilon)\})$, so:

$$\begin{aligned} &= \left\{ w \mid (\tau, w) \in \underbrace{\llbracket P_1; \dots; P_1 \rrbracket}_{\sigma(x) \text{ times}}(\{(\sigma, \varepsilon)\}) \right\} \quad (\text{D.3.8}) \\ &= L_{S_\varepsilon}(\underbrace{\llbracket P_1; \dots; P_1 \rrbracket}_{\sigma(x) \text{ times}}(\{(\sigma, \varepsilon)\})), \end{aligned}$$

which resembles the case for the concatenation.

ω -Loops

$$\begin{aligned} & L_{S_\varepsilon}(\text{LOOP } \omega \text{ DO } P_1 \text{ END}) \\ &= \{w \mid (\tau, w) \in \llbracket \text{LOOP } \omega \text{ DO } P_1 \text{ END} \rrbracket(\{(\sigma, \varepsilon)\})\} \end{aligned} \quad (\text{D.3.10})$$

$$\begin{aligned} &= \left\{ w \mid (\tau, w) \in \left\{ (\perp, v) \mid \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \llbracket P_1 \rrbracket^k(\{(\sigma, \varepsilon)\}) \end{array} \right\} \right\} \quad (\text{D.3.8}) \end{aligned}$$

$$= \left\{ v \mid \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_0 \cdot v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_0 \cdot v_1 \cdots v_k) \in \llbracket P_1 \rrbracket^k(\{(\sigma, \varepsilon)\}) \end{array} \right\}$$

Since v_0 is inevitably equal to ε , we may simplify this to:

$$= \left\{ v \mid \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_1 \cdots v_k) \in \llbracket P_1 \rrbracket^k(\{(\sigma, \varepsilon)\}) \end{array} \right\}$$

$$\begin{aligned}
&= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N} \exists \sigma, \\ \text{such that } (\sigma, v_1 \cdots v_k) \in \underbrace{[[P_1; \dots; P_1]]}_{k \text{ times}}(\{(\sigma, \varepsilon)\}) \end{array} \right. \right\} \\
&= \left\{ v \left| \begin{array}{l} v \in \Sigma_{I/O}^\omega, \exists v_1 \cdot v_2 \cdots = v, \\ \text{such that } v_i \in \Sigma_{I/O}^* \text{ and } \forall k \in \mathbb{N}, \\ \text{such that } v_1 \cdots v_k \in L \left(\underbrace{P_1; \dots; P_1}_{k \text{ times}} \right) \end{array} \right. \right\} \tag{D.3.10}
\end{aligned}$$

As we can see, there are arbitrarily many different prefixes of the words v in the language $L_{S_\varepsilon}(\text{LOOP } \omega \text{ DO } P_1 \text{ END})$ which are all in a language of the form

$$L \left(\underbrace{P_1; \dots; P_1}_{k \text{ times}} \right).$$

By means of the case for concatenation, all languages of that form are reactive and thus $L_{S_\varepsilon}(\text{LOOP } \omega \text{ DO } P_1 \text{ END})$ is reactive as well. This concludes the proof of Lemma L.4.5. \square

A.3 Alternative Proof of Lemma L.5.6

Lemma L.5.6 (UNDECIDABILITY OF THE REGULARITY PROBLEM FOR Loop_ω):
The Regularity Problem for Loop_ω is undecidable.

We will prove Lemma L.5.6 by reducing the Halting Problem for WHILE programs to the Regularity Problem for Loop_ω . For the reduction we need to construct out of a WHILE program P a $\text{LOOP}+\omega$ program whose language's regularity depends solely on the halting behavior of P . From any WHILE program P in Kleene Normal Form we can construct such a $\text{LOOP}+\omega$ program O'_P (a slightly modified version of the $\text{LOOP}+\omega$ simulation of P) as follows:

Definition D.A.1 (MODIFIED $\text{LOOP}+\omega$ SIMULATION OF WHILE PROGRAMS):

Let P be a WHILE program in Kleene Normal Form (otherwise, transform P into Kleene Normal Form first), i.e.

$$P = x_k := x_k + 1; \text{ WHILE } x_k \neq 0 \text{ DO } P' \text{ END.}$$

Then the modified $\text{LOOP}+\omega$ simulation O'_P of P is the following $\text{LOOP}+\omega$ program:

```

1:  $x_k := 1;$ 
2:  $x_j := 1;$ 
3: LOOP  $\omega$  DO
4:     IF  $x_k \neq 0$  THEN
5:          $P'$ ;

```

```

6:          LOOP  $x_j$  DO
7:              sendJ a;
8:          END;
9:           $x_j := x_j + 1$ 
10:         ELSE
11:             skip
12:         END;
13:         pulse
14:     END

```

For this modified LOOP+ ω simulation we can now prove that its language depends solely on the halting behavior of the simulated WHILE program. For that, we first examine which language is generated in case of termination of P and which language is generated in case of non-termination of P :

Lemma L.A.4:

The following holds for the language $L(O'_P)$ of the modified LOOP+ ω simulation O'_P of a WHILE program P :

$$L(O'_P) = \begin{cases} \prod_{i=1}^{\infty} \{ \langle I \xrightarrow{a} J \rangle^i \cdot \underbrace{\quad}_I \}, & \text{if } P \text{ does not halt} \\ \prod_{i=1}^c \{ \langle I \xrightarrow{a} J \rangle^i \cdot \underbrace{\quad}_I \} \cdot \{ \underbrace{\quad}_I \}^\omega, & \text{if } P \text{ halts} \end{cases}$$

where $c \in \mathbb{N}$ is the number of iterations, the loop body is executed before P halts, if P halts.

Proof: Assume P halts. That means that P eventually leaves its single while-loop which is only the case if within finitely many executions of the loop body P' the variable x_k is set to 0 (recall that P is in Kleene Normal Form). If this is the case, then in the execution of O'_P the variable x_k is also set to 0 after finitely many executions of the ω -loop body. As long as the variable evaluates to something other than 0, after each execution of P' the sequence $\langle I \xrightarrow{a} J \rangle^i \cdot \underbrace{\quad}_I$, where i is the number of times P' has already been executed, is generated. As soon as x_k is set to zero after c many executions of P' , the variable x_k is never again touched and the $\langle I \xrightarrow{a} J \rangle^i \cdot \underbrace{\quad}_I$ sequences are never again generated. Only the pulse symbol is generated from here on ad infinitum. Hence the language of O'_P is in this case given by

$$\underbrace{\prod_{i=1}^c \{ \langle I \xrightarrow{a} J \rangle^i \cdot \underbrace{\quad}_I \}}_{x_k \text{ not yet 0}} \cdot \underbrace{\{ \underbrace{\quad}_I \}^\omega}_{x_k \text{ is 0 after } c \text{ executions of } P'}$$

Now assume P does not halt. That means that P never leaves its single while-loop which is only the case if P' will be executed infinitely many times without ever setting x_k to 0. If this is the case, then in the execution of O'_P the variable x_k is also never set to 0. As long as the variable evaluates to something other than 0, the

$\{I \xrightarrow{a} J\}^i \cdot \underbrace{\quad}_I$ sequences are generated. Since this is infinitely often the case, as P does not halt, infinitely many $\{I \xrightarrow{a} J\}^i \cdot \underbrace{\quad}_I$ sequences with increasing i are generated. Hence the language of O'_P is in this case given by

$$\prod_{i=1}^{\infty} \left\{ \{I \xrightarrow{a} J\}^i \cdot \underbrace{\quad}_I \right\}. \quad \square$$

For proving that the regularity of the language of the modified LOOP+ ω simulation depends solely on the halting behavior of P , it remains to show that $L(O'_P)$ is regular, if and only if P halts:

Lemma L.A.5:

$L(O'_P)$ is regular if and only if P halts.

Proof: Assume P halts. Then by means of Lemma L.A.4 we know that

$$L(O'_P) = \prod_{i=1}^c \left\{ \{I \xrightarrow{a} J\}^i \cdot \underbrace{\quad}_I \right\} \cdot \left\{ \underbrace{\quad}_I \right\}^{\omega}.$$

The language $\prod_{i=1}^c \left\{ \{I \xrightarrow{a} J\}^i \cdot \underbrace{\quad}_I \right\}$ is finite, as $c \in \mathbb{N}$ is a constant, and therefore regular as every finite language is regular (cf. Lemma L.2.1). The language $\left\{ \underbrace{\quad}_I \right\}^{\omega}$ is obviously ω -regular. As ω -regular languages are closed under left-concatenation with regular languages (cf. Remark R.2.10) the language $L(O'_P)$ is ω -regular.

Now assume P does not halt. Then by means of Lemma L.A.4 we know that

$$L(O'_P) = \prod_{i=1}^{\infty} \left\{ \{I \xrightarrow{a} J\}^i \cdot \underbrace{\quad}_I \right\}.$$

Note that in this case $L(O'_P)$ is a finite ω -language containing exactly one ω -word. However, $L(O'_P)$ is not regular: According to [CG77b] it holds that if $L(O'_P)$ is ω -regular, then the set $\text{Pref}(L(O'_P))$ of all prefixes of the words in $L(O'_P)$ has to be regular. The set $\text{Pref}(L(O'_P))$ is given by

$$\text{Pref}(L(O'_P)) = \left\{ \left(\prod_{i=1}^k \left\{ \{I \xrightarrow{a} J\}^i \cdot \underbrace{\quad}_I \right\} \right) \cdot \{I \xrightarrow{a} J\}^j \mid k \in \mathbb{N}, j \leq k+1 \right\}.$$

If this language was regular, then, according to the pumping lemma for regular languages (cf. Remark R.2.3), there exists a constant $n \in \mathbb{N}$ such that for every word $w \in \text{Pref}(L(O'_P))$ with $|w| \geq n$ there exists a decomposition $w = vxz$ such that $|vx| \leq n$, $x \neq \varepsilon$ and for all $\ell \in \mathbb{N}$ it holds that $vx^{\ell}z \in \text{Pref}(L(O'_P))$.

Now consider any arbitrary word $w \in \text{Pref}(L(O'_P))$ with $|w| \geq n$. For any decomposition $w = vxz$ with $x \neq \varepsilon$ and for any $\ell \neq 1$ it obviously holds that $vx^{\ell}z \notin \text{Pref}(L(O'_P))$, because almost any modification (except adding a *limited* number of $\{I \xrightarrow{a} J\}$ symbols at the end of the word) causes the word not to be in the language anymore. Hence $\text{Pref}(L(O'_P))$ does not respect the pumping lemma and therefore cannot be regular. Thus, by means of [CG77b], the language $L(O'_P)$ cannot be ω -regular. \square

With $L(O'_P)$ we have a language at hand, whose regularity depends solely on the halting behavior of the WHILE program P . We can now finally prove Lemma L.5.6:

Proof of Lemma L.5.6: Assume the Regularity Problem was decidable for $\mathcal{L}oop_\omega$. Then there existed a computable algorithm \mathfrak{R} which decides whether the language of any given LOOP+ ω program is regular. With \mathfrak{R} at hand, we construct an algorithm \mathfrak{H} which decides the Halting Problem for WHILE programs, using the algorithm \mathfrak{R} as a subprogram.

Description of \mathfrak{H} : Given a WHILE program P , where P is w.l.o.g. in Kleene Normal Form. Then do the following:

1. Construct the modified LOOP+ ω simulation O'_P of P .
2. Pass O'_P to \mathfrak{R} .
- 3a. If $\mathfrak{R}(O'_P)$ returns “Yes” (i.e. $L(O'_P)$ is regular), then P halts, so return “Yes”.
- 3b. If $\mathfrak{R}(O'_P)$ returns “No” (i.e. $L(O'_P)$ is not regular), then P does not halt, so return “No”.

Partial correctness of \mathfrak{H} :

$$\begin{aligned}
 P \text{ halts} &\implies L(O'_P) \text{ is regular} && \text{(L.A.5)} \\
 &\implies \mathfrak{R}(O'_P) = \text{“Yes”} && (\mathfrak{R} \text{ decides Regularity Problem)} \\
 &\implies \mathfrak{H}(O'_P) = \text{“Yes”} && \text{(Step 3a of } \mathfrak{H}\text{)}
 \end{aligned}$$

$$\begin{aligned}
 P \text{ does not halt} &\implies L(O'_P) \text{ is not regular} && \text{(L.A.5)} \\
 &\implies \mathfrak{R}(O'_P) = \text{“No”} && (\mathfrak{R} \text{ decides Regularity Problem)} \\
 &\implies \mathfrak{H}(O'_P) = \text{“No”} && \text{(Step 3b of } \mathfrak{H}\text{)}
 \end{aligned}$$

Total correctness of \mathfrak{H} : All steps of \mathfrak{H} which do not invoke \mathfrak{R} are obviously computable and terminate. By assumption the invocation of \mathfrak{R} is also computable and terminates. Hence \mathfrak{H} is computable and terminates.

The algorithm \mathfrak{H} is hence a correct algorithm which decides the Halting Problem, which is a contradiction to the undecidability of the Halting Problem. Therefore \mathfrak{R} cannot exist and thus the Regularity Problem is undecidable. \square

Nomenclature

L^*	Kleene-closure of the language L , page 6
L^+	$L \cdot L^*$, the positive closure of the language L , page 6
L^ω	ω -closure of the language L , page 13
$w[i]$	The i^{th} symbol of the word w , page 5
$w _\Sigma$	Projection of the word w on the alphabet Σ , page 82
$\wr J \xrightarrow{x} I $	Symbol that indicates that program P_J sends symbol x to program P_I , page 24
$ J \xrightarrow{x} I \wr$	Symbol that indicates that program P_I reads symbol x from channel $C_{J \rightarrow I}$, page 24
$ J \xrightarrow{\perp} I \wr$	Symbol that indicates that Program P_I is attempting to read from the empty channel $C_{J \rightarrow J}$, page 24
$\sim \sim \sim^I$	Symbol that indicates that program P_I completed one iteration of its ω -loop, page 24
\cdot	Concatenation of two words or languages, page 5
\boxtimes	The synchronized product for finite automata, page 82
\boxtimes_ω	The synchronized product for Büchi Automata, page 84
\otimes	The synchronization of two languages, page 82
$\dot{\cup}$	Disjoint union, page 43
$\langle P \rangle$	Semantics of a LOOP program P , page 20
$\llbracket P \rrbracket$	Semantics of a LOOP+ ω program or a LOOP program with message passing P , page 29
$;$	Concatenation of two programs, page 20
$\mathfrak{A}_{J \rightarrow I}^k$	The channel guard automaton for channel $C_{J \rightarrow I}$ with channel size k , page 89
\mathcal{CFL}	The class of context-free languages, page 11

\mathcal{CFL}_ω	The class of ω -context-free languages, page 16
$C_{J \rightarrow I}$	The channel located at program P_I dedicated to receiving and queuing messages from program P_J , page 22
\mathcal{CSL}	The class of context-sensitive languages, page 11
DBA	Deterministic Büchi Automaton, page 14
DFA	Deterministic finite automaton, page 6
$\text{dom}(P)$	The domain of the function calculated by the WHILE program P , page 52
ε	Empty word, page 5
$\gamma(\cdot)$	Returns the Gödel Number of its argument, page 48
GBA	Generalized Büchi Automaton, page 84
$L(\mathfrak{A})$	Language recognized by the automaton \mathfrak{A} , page 7
$L(P)$	Language of a program P , page 30
$L_S(P)$	Language generated by a program P when executed on the initial set of configurations S , page 30
$L(S)$	Language of a set of configurations S , page 30
\mathbb{L}	The set of all LOOP programs, page 20
\mathbb{L}_ω	The set of all LOOP+ ω programs, page 29
$\widehat{L}(P)$	The natural over-approximation of the language of the program P , page 71
$\lim L$	Limit of the language L , page 13
$L_{J \rightarrow I}^k$	The channel guard language for channel $C_{J \rightarrow I}$ with channel size k , page 90
LOOP	The model programming language LOOP, page 20
LOOP+ ω	The model programming language LOOP+ ω , page 29
\mathcal{Loop}_ω	The class of LOOP+ ω -generated languages, page 30
\mathbb{N}	The set of natural numbers including 0, page 6
NBA	Non-deterministic Büchi Automaton, page 15
NFA	Non-deterministic finite automaton, page 8
ω	The cardinality of the natural numbers, page 12
O_P	The LOOP+ ω simulation of the WHILE program P , page 50
$\mathcal{P}(\cdot)$	Powerset operator, page 5

$\text{pdb}(\cdot)$	Returns the pulse distance bound of its argument, page 66
$\text{Pref}(w)$	The set of all prefixes of the word w , page 13
\mathcal{RE}	The class of recursively enumerable languages, page 12
\mathcal{RE}_ω	The class of ω -recursive languages, page 17
\mathcal{Reg}	The class of regular Languages, page 6
\mathcal{Reg}_ω	The class of ω -regular languages, page 13
Σ	Finite alphabet, page 5
Σ^*	Set of all words over the alphabet Σ , page 5
Σ^ω	Set of all ω -words over the alphabet Σ , page 13
Σ^∞	$\Sigma^* \cup \Sigma^\omega$, page 13
S_0	Abbreviation for $\{(\sigma_0, \varepsilon)\}$, page 30
$\sigma_{(v_0, \dots, v_k)}$	A valuation function where the first k variables are set to v_0, \dots, v_k respectively, page 20
σ_0	A valuation function in which every variable evaluates to 0, page 30
\mathbb{S}	Set of all configurations which can emerge from executing a LOOP program, page 29
\mathbb{S}_\perp	Set of all configurations which can emerge from executing a LOOP program or a LOOP+ ω program, page 29
$\Sigma_{I/O}$	The alphabet of the I/O-languages generated by the LOOP+ ω programs, page 24
$\text{strip}(\cdot)$	Function that removes all leading pulse symbols from its argument, page 44
\mathcal{Var}	The set of all program variables, page 19
\mathbb{W}	The set of all WHILE programs, page 47
WHILE	The model programming language WHILE , page 47

Bibliography

- [Ack28] Wilhelm Friedrich Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99(1):118 – 133, 1928.
- [Büc60] Julius Richard Büchi. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly*, 6(1-6):66 – 92, 1960.
- [Büc66] Julius Richard Büchi. Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic. In Patrick Suppes Ernest Nagel and Alfred Tarski, editors, *Logic, Methodology and Philosophy of Science Proceeding of the 1960 International Congress*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1 – 11. Elsevier, 1966.
- [BZ83] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323 – 342, April 1983.
- [CG77a] Rina S. Cohen and Arie Y. Gold. Theory of ω -Languages. I: Characterizations of ω -Context-Free Languages. *Journal of Computer and System Sciences*, 15(2):169 – 184, 1977.
- [CG77b] Rina S. Cohen and Arie Y. Gold. Theory of ω -Languages. II: A Study of Various Models of ω -Type Generation and Recognition. *Journal of Computer and System Sciences*, 15(2):185 – 208, 1977.
- [CG78a] Rina S. Cohen and Arie Y. Gold. ω -Computations on Deterministic Push-down Machines. *Journal of Computer and System Sciences*, 16(3):275 – 300, 1978.
- [CG78b] Rina S. Cohen and Arie Y. Gold. ω -Computations on Turing Machines. *Theoretical Computer Science*, 6(1):1 – 23, 1978.
- [DSW94] Martin D. Davis, Ron Sigal, and Elaine J. Wyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Elsevier Academic, 1994.
- [HMU02] John Edward Hopcroft, Rajeev Motwani, and Jeffrey David Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison Wesley, 2002.
- [Hoa78] Charles Antony Richard Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666 – 677, August 1978.

- [Kle36] Stephen Cole Kleene. General Recursive Functions of Natural Numbers. *Mathematische Annalen*, 112(1):727–742, 1936.
- [Kle38] Stephen Cole Kleene. On Notation for Ordinal Numbers. *The Journal of Symbolic Logic*, 3(4):pp. 150–155, 1938.
- [Kle43] Stephen Cole Kleene. Recursive Predicates and Quantifiers. *Transactions of the American Mathematical Society*, 53(1):41 – 73, 1943.
- [Kle56] Stephen Cole Kleene. Representation of Events in Nerve Nets and Finite Automata. In Claude Elwood Shannon and John McCarthy, editors, *Automata Studies*, pages 3 – 42. Princeton University Press, 1956.
- [Lan69] Lawrence Hugh Landweber. Decision Problems for ω -Automata. *Mathematical Systems Theory*, 3(4), 1969.
- [MR67] Albert Ronald da Silva Meyer and Dennis MacAlistair Ritchie. The Complexity of Loop Programs. In *Proceedings of the 1967 22nd National Conference*, ACM '67, pages 465 – 469, New York, NY, USA, 1967. ACM.
- [MS72] Albert Ronald da Silva Meyer and Larry Joseph Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *Switching and Automata Theory, 1972., IEEE Conference Record of 13th Annual Symposium on*, pages 125–129, 1972.
- [Myh57] John R. Myhill. Finite Automata and the Representation of Events. Technical Report WADC TR-57-624, Wright-Paterson Air Force Base, 1957.
- [Ner58] Anil Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541 – 544, 1958.
- [Pot94] Andreas Potthoff. *Logische Klassifizierung regulärer Baumsprachen*. PhD thesis, Christian-Albrechts-Universität, Institut für Informatik und Praktische Mathematik, 1994.
- [Ric53] Henry Gordon Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358 – 366, 1953.
- [Ric08] Elaine Rich. *Automata, Computability, and Complexity: Theory and Applications*. Pearson Prentice Hall, 2008.
- [RS59] Michael Oser Rabin and Dana Stewart Scott. Finite Automata and their Decision Problems. *IBM Journal of Research and Development*, 3(2):114 – 125, 1959.
- [Sch65] Marcel-Paul Schützenberger. On Finite Monoids Having Only Trivial Subgroups. *Information and Control*, 8(2):190 – 194, 1965.
- [Sch00] Uwe Schöning. *Theoretische Informatik — kurz gefasst*. Spektrum Akademischer Verlag, Heidelberg Berlin, 2000.

- [SDK⁺11] Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Klaus Wehrle, Carsten Weise, and Stefan Kowalewski. Scalable Symbolic Execution of Distributed Systems. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 333–342, 2011.
- [SLA⁺10] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN '10*, pages 186–196, New York, NY, USA, 2010. ACM.
- [Sta97] Ludwig Staiger. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages, Vol. 3 Beyond Words*, chapter ω -Languages, pages 339 – 387. Springer-Verlag New York, Inc., 1997.
- [Tho90] Wolfgang Thomas. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, chapter Automata on Infinite Objects, pages 133 – 191. MIT Press, 1990.
- [Tur36] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230 – 265, 1936.
- [VC83] Son T. Vuong and Donald D. Cowan. Reachability Analysis of Protocols with FIFO Channels. *SIGCOMM Comput. Commun. Rev.*, 13(2):49 – 57, April 1983.
- [VW84] Moshe Ya'akov Vardi and Pierre Wolper. Automata Theoretic Techniques for Modal Logics of Programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, STOC '84*, pages 446 – 456, New York, NY, USA, 1984. ACM.