

# Control Conditions for Transformation Units – Parallelism, As-long-as-possible, and Stepwise Control

von Melanie Luderer

DISSERTATION

zur Erlangung des Grades eines Doktors der  
Ingenieurwissenschaften  
- Dr.Ing. -

vorgelegt im Fachbereich 3 (Mathematik und Informatik) der  
Universität Bremen  
Dezember 2015

Gutachter: Prof. Dr. Hans-Jörg Kreowski  
Prof. Dr. Julia Padberg



## Danksagung

Von ganzem Herzen möchte ich Hans-Jörg Kreowski und der Arbeitsgruppe Theoretische Informatik (Larbi Abdenebaoui, Marcus Ermler, Sabine Kuske und Caroline von Totth) danken. Durch sowohl fachliche als auch persönliche Gespräche habe ich großartige Unterstützung erhalten. Nicht minder gilt mein Dank meiner Mutter, die mich tatkräftig unterstützt und bestärkt hat. Zuguterletzt möchte ich mich bei International Graduate School for Dynamics in Logistics (IGS) bedanken, die mich zum einen durch ein Stipendium finanziell unterstützt hat und zum anderen mit zahlreichen Angeboten die Ausarbeitung des Themas und die Anfertigung der Dissertation begleitet hat.



## **Abstract**

The concept of graph transformation units is a formal and as well intuitive means to model processes on graphs. Thereby the control condition of a transformation unit plays an important role. It provides so to say the intelligence of the unit by describing its desired behaviour. The thesis regards control conditions focusing on two aspects: expressivity and practicability. Considering expressivity it implements two kinds of control conditions, as-long-as-possible and parallel expressions. As their names imply these control conditions are able to express the as-long-as-possible iteration respectively parallel composition of already described behaviour. Focusing on practical executability the thesis introduces the concept of stepwise control conditions. Whereas conventional control conditions in principle describe desired behaviour their computation may take a long time, since first all possible derivations have to be computed and then are checked against the control condition. Stepwise control conditions allow to directly guide the derivation process and so may save computation time.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Preliminaries</b>	<b>5</b>
2.1	Formal Languages . . . . .	5
2.1.1	Regular Expressions and Regular Languages . . . . .	7
2.2	Finite State Automata . . . . .	8
2.2.1	Composition of Automata . . . . .	10
2.2.2	Transform Regular Expressions to Finite State Automata	11
2.3	Graphs and Rule-based Graph Transformation . . . . .	12
2.3.1	Graphs and Operations Regarding Graphs . . . . .	12
2.3.2	Rule-based Graph Transformation . . . . .	14
2.3.3	Graph Transformation Units . . . . .	17
2.3.4	Parallel Graph Transformation Rules . . . . .	20
2.4	Petri Nets . . . . .	20
<b>3</b>	<b>Parallel Control Conditions</b>	<b>24</b>
3.1	Parallelism and Graph Transformation . . . . .	25
3.2	Parallel Expressions . . . . .	27
3.2.1	Language of Weak Parallel and Synchronous Expressions . . . . .	28
3.2.2	Properties of Weak and Synchronous Composition . . . . .	32
3.2.3	Algebraic Laws for Weak and Synchronous Expressions	39
3.2.4	Weak and Synchronous Expressions as Control Conditions . . . . .	44
<b>4</b>	<b>Languages of Parallel Expressions are Regular</b>	<b>57</b>
4.1	Parallel Expressions to Automata . . . . .	57
4.2	Automata Recognise Parallel Languages . . . . .	62
<b>5</b>	<b>As-long-as-possible Control Condition</b>	<b>73</b>
5.1	As-long-as-possible Expressions . . . . .	73

5.2	Syntax and Semantics . . . . .	74
5.3	Further Notions and Definitions . . . . .	76
<b>6</b>	<b>Sufficient Conditions for Termination of As-long-as-possible Expressions</b>	<b>78</b>
6.1	Termination in Literature . . . . .	79
6.2	Termination Regarding <i>Alap</i> -expressions . . . . .	80
6.2.1	Strong Termination . . . . .	81
6.2.2	Structural Termination . . . . .	83
6.2.3	Assured Termination . . . . .	86
6.3	An Algorithm to Check Assured Termination . . . . .	94
6.3.1	Example . . . . .	94
<b>7</b>	<b>Stepwise Controls</b>	<b>97</b>
7.1	Basic Stepwise Controls . . . . .	98
7.1.1	Definition and Construction . . . . .	98
7.1.2	Execution of Stepwise Controls . . . . .	101
7.1.3	Semantics of Stepwise Controls . . . . .	102
7.2	Transform Given Control Conditions to Stepwise Controls . . . . .	103
7.2.1	Weak and Synchronous Stepwise Controls . . . . .	103
7.2.2	Parallel Stepwise Controls . . . . .	106
7.2.3	As-long-as-possible Stepwise Controls . . . . .	113
7.3	Transformation Units and Stepwise Controls . . . . .	118
7.3.1	Stepwise Controls for Transformation Units . . . . .	118
7.3.2	Transformation Unit as Stepwise Control . . . . .	120
<b>8</b>	<b>Conclusion</b>	<b>122</b>



# Chapter 1

## Introduction

In computer science modelling plays an important role. Think of, e.g., UML diagrams, Petri nets, event driven process chains, and last but not least the von Neumann model of the first computer. In general, models are designed to represent aspects of the real world or ideas. They abstract from irrelevant details and help to grasp and understand the modelled system and its behaviour. One major application using models is to simulate, respectively analyse, the behaviour of the modelled system in order to gain information. Simulation could be employed to validate hypotheses about the modelled system or to forecast its behaviour, e.g. weather forecast. Another field of application regarding simulation is problem solving. Thereby the problem domain and potential ways to the solution of the problem are modelled, then the model is run in order to gain possible solutions.

### **Graph transformation units as a modelling framework**

Models are implemented in the language of a modelling framework. This framework could be formal or informal. An advantage of formal frameworks is that they provide means to exactly analyse the modelled system and processes, validate hypotheses, and prove properties. Also the verification of the model is, if at all, rather possible in formal than in informal frameworks. On the other hand formal frameworks often lack intuitive usability. To recognise the world in a pure mathematical model is rather difficult for most non-mathematicians. The concept of graph transformation units, as studied in [Kus00] is a formal modelling framework, which combines the just mentioned advantages of formality with the ability to intuitively and clearly model rather complex processes and systems. Transformation units encapsulate graph transformation rules, a control condition to regulate the application of these rules, and graph class expressions specifying permitted

initial and terminal graphs. Hence, transformation units can be seen as entities performing specific tasks in an environment modelled by a graph. Once implemented they are (re)usable to accomplish their tasks without the need to know how exactly this is done. The structuring principle of import enables a unit to employ other units. This principle helps to structure the model and keeps it handable.

### **Control conditions**

The thesis focuses on control conditions as a core element providing the "intelligence" of transformation units. The control condition regulates the application of the unit rules. Examples for control conditions for transformation units can be found in, e.g., [Kus98, KK99]. Without a control condition a transformation unit could only non-deterministically choose rules to apply from its given rule set. From the unit's point of view this behaviour could be characterised as blind search. It arbitrarily tries rule combinations and after each application checks if coincidentally a desired state is reached. If some kind of regulation is wanted it has to be encoded in the rules, which can make them quite incomprehensible. Control conditions equip the transformation unit with the ability to follow strategies and thus behave in a problem-specific way. The thesis considers control conditions regarding two aspects: expressivity and practicability. The expressivity of a control condition determines the potential the modeller has to describe a specific behaviour. The more means of expression control conditions have the more "intelligent" and problem-specific behaviour they can describe. The practicability of a control condition regards the time it takes to actually compute the behaviour of the respective unit.

Regarding the expressivity the thesis introduces two kinds of control conditions, *parallel expressions* and *as-long-as-possible expressions*. Both are constructed by enhancing regular expressions by additional operators.

### **Parallel expressions**

The ability to model parallelism is crucial since parallelism is an inherent property of the real world. Parallelism in graph transformation is, e.g., focused in [Kre78, Roz97]. In addition parallel execution saves time. For example consider the Travelling Salesperson Problem. The computation of the shortest route takes less time when different possibilities can be explored simultaneously. For instance, in [KK11] it was shown that for graph multiset transformation, that is graph transformation running not only on a single input graph, but on a multiset of graphs, "NP-problems with graph-transformational solutions can be solved [...] in a polynomial number of steps

with arbitrarily high probability”. Hence parallel execution of steps can reduce the time complexity from exponential to polynomial. The ability to express parallelism with graph transformation on rule level is given by parallel graph productions, also called parallel rules. A parallel rule composes two given rules to a new one. In e.g. [KK07] parallel rules were used to model the simultaneous activity of the members of a community of transformation units. However, the units themselves are not able to directly use parallelism in their control conditions. In order to equip control conditions with the ability to express parallelism, we do not only have to consider the parallel composition of rules, but also of rule sequences. On the level of rule sequences though, parallel composition is ambiguous. Therefore in the thesis we distinguish three different forms of parallel composition: weak, proactive, and synchronous composition. Regarding these different forms the thesis introduces weak parallel, proactive, and synchronous expressions, which provide the ability to compose entire expressions in the respective parallel form. All three expressions are implemented by enhancing regular expressions by a respective additional parallel composition operator. As an interesting result it turns out that the languages of weak and synchronous expressions are still regular.

### **As-long-as-possible expressions**

When modelling problem solving strategies the ability to do something as long as possible is very convenient. For instance, consider the subtask to delete all  $x$ -labelled edges in a graph. Without the ability to express the execution the  $x$ -label-deleting rule as long as possible one would have to e.g. encapsulate the desired behaviour in a transformation unit. Thereby the encapsulating unit iterates the deleting rule arbitrarily often and its terminal graph class expression allows only those graphs where no more deletion is possible. On the one hand such a proceeding only is possible when employing structured transformation units. On the other hand to be forced to model such a subtask by a separate transformation unit could complicate the structuring desired by employing units with import. As-long-as-possible control conditions were addressed in, e.g., [Kus00, Kus98, KKS97, BHPT05]. The first two implement as-long-as-possible control conditions on a single rule respectively on a set of rules iterating the rule (respectively rules) as long as no more application is possible. The third models as-long-as-possible on arbitrary, not specified control conditions, providing a binary semantic relation on graphs. The latter provides as-long-as-possible for expressions containing in addition sequential and choice compositions. The thesis studies so called as-long-as-possible expressions, which enhance regular expressions by a binary as-long-as-possible operator, denoted by  $!$ . The operator  $!$  then provides

the iterative application of its argument as long as it is applicable. As shown in [HKK06], as-long-as-possible control conditions in general are not decidable (not even semi-decidable), hence also termination of as-long-as-possible expressions is not decidable. Considering this situation the thesis makes use of the formality of graph transformation and introduces an approach to analyse as-long-as-possible expressions regarding sufficient conditions for termination.

Regarding practicability one finds that several formal frameworks lack attention on practicability. Also the definition of the semantics of transformation units given in the preliminaries is not very practicable. It is defined by building arbitrarily derivations and then check them against the control condition. This proceeding is not very efficient since many derivations are build which never have a chance to be permitted by the control condition.

### **Stepwise control**

In order to increase the practicability of control conditions we introduce *stepwise control conditions*. Stepwise controls are similar to finite state automata, but enhanced with components which allow more control over the derivation process and to take into account the current graph. In contrast to many control conditions, which are used to check already computed derivations, stepwise controls directly build only those derivations which may end up to be permitted. Thus stepwise controls reduce the time it takes to execute a transformation unit. Being able to take into account the current graph stepwise controls are also well suited to model proactive parallelism.

### **Structure of the thesis**

The next chapter contains preliminary definitions and notations used throughout the thesis. Chapter 3 introduces parallel expressions and in particular weak parallel and synchronous expressions. Chapter 4 then states and proves regularity of languages provided by weak parallel and synchronous expressions. In Chapter 5 as-long-as-possible expressions are introduced followed by Chapter 6 introducing sufficient conditions for termination of as-long-as-possible expressions. Chapter 7 then introduces stepwise controls. The thesis concludes with a summary of the presented work and some suggestions for future work.

# Chapter 2

## Preliminaries

The Preliminaries contain basic definitions and notations used throughout the thesis. The chapter comprises four sections regarding formal languages, automata theory, basics of graph transformation, and Petri nets. These sections do not give an overview over the respective area, but recall the concepts and notations as we use them in the thesis. For the reader interested in further information, each section gives references to books or articles where more detailed information can be found.

### 2.1 Formal Languages

In theoretical computer science a *language* is a set of *words*, which themselves again are sequences of symbols. A collection of symbols is called *alphabet*. The following definitions express this formally and also provide some operations on words and languages. The definitions base on [KK14].

#### Definition 1. Alphabets and words

1. An *alphabet* is a set of symbols.  
E.g.,  $\{0, 1\}$  or  $\{a, b, \dots, z\}$ .
2. A *word* over an alphabet  $\Sigma$  is a finite sequence of symbols from  $\Sigma$ .  
E.g., 1011 or *ab*.
3. The *empty word*, i.e. the word which contains no symbols, is denoted by  $\lambda$ .
4. A word over an alphabet  $\Sigma$  is recursively defined by:
  - (i)  $\lambda$  is a word,
  - (ii) given  $x \in \Sigma$  and a word  $v$  then also  $xv$  is a word.

The last construction step (4.(ii)) is also called *left addition*.

Given two words one can compose a new one by their juxtaposition. This operation is called *concatenation*.

**Definition 2. Concatenation of words**

Let  $v$  and  $w$  be two words over an alphabet  $\Sigma$ . The *concatenation* of  $v$  and  $w$ ,  $v \circ w$ , is recursively defined by:

- (i) given  $v = \lambda$  it holds  $\lambda \circ w = w$ ,
- (ii) given  $v = xu$  with  $x \in \Sigma$  it hold  $(xu) \circ w = x(u \circ w)$ .

Often the operator  $\circ$  is omitted, i.e. one can write  $vw$  instead of  $v \circ w$ .

A language is a set of words. Formally it is defined as a subset of the set of all (possible) words over a given alphabet.

**Definition 3.  $\Sigma^*$  and language**

Let  $\Sigma$  be an alphabet. The set of all words over  $\Sigma$  is denoted by  $\Sigma^*$  and recursively defined by

$$\Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k \text{ with}$$

- (i)  $\Sigma^0 = \{\lambda\}$ ,
- (ii)  $\Sigma^{i+1} = \{x \circ u \mid x \in \Sigma, u \in \Sigma^i\}, i \in \mathbb{N}$ .

A *language* is a subset of  $\Sigma^*$ , e.g.,  $\{i, am, a, language\} \subseteq \{a, b, \dots, z\}^*$ .

Analogously to words one can also concatenate two languages  $L_1$  and  $L_2$ . The resulting language then consists of words the first part of which comes from  $L_1$  and the second part from  $L_2$ .

**Definition 4. Concatenation of two languages**

Let  $L_1$  and  $L_2$  be two languages. Then

$$L_1 \circ L_2 = \{x_1 \circ x_2 \mid x_1 \in L_1, x_2 \in L_2\}.$$

Also the operator  $\circ$  for languages often is omitted writing  $L_1L_2$  instead of  $L_1 \circ L_2$ .

A further analogous operator on language level is the iteration of a language. Given a language  $L$  the iteration  $L^*$  builds the set of all words constructed by concatenation of arbitrary many words from  $L$ .

**Definition 5. Iteration of a language**

Let  $L$  be a language. Then

$$L^* = \bigcup_{n \in \mathbb{N}} L^n \text{ with}$$

$$L^0 = \{\lambda\},$$

$$L^1 = L,$$

$$L^n = L \circ L^{n-1}, n > 1.$$

Since languages are sets all operations available for sets can also be applied to languages.

**2.1.1 Regular Expressions and Regular Languages**

Regular expressions describe specific languages, called *regular languages*. The following definitions address regular expressions and how a regular expression describes a language. The definitions are taken from [Sch92].

**Definition 6. Regular expressions**

Let  $\Sigma$  be an alphabet. Regular expressions over  $\Sigma$  are recursively defined by:

- $\emptyset$  and  $\lambda$  are regular expressions,
- $\sigma \in \Sigma$  is a regular expression,
- $(e_1; e_2), (e_1|e_2)$  are regular expressions if  $e_1$  and  $e_2$  are regular expressions, and
- $e^*$  is a regular expression if  $e$  is a regular expression.

The binding strength of the operators is given by  $* > > |$ .

Every regular expression  $e$  over an alphabet  $\Sigma$  describes a language  $L(e) \subseteq \Sigma^*$ .

**Definition 7. Regular language**

Let  $\Sigma$  be an alphabet. The language of a regular expression over  $\Sigma$  is recursively defined by

- $L(\emptyset) = \emptyset$ ,
- $L(\lambda) = \{\lambda\}$ ,
- $L(\sigma) = \{\sigma\}$  for  $\sigma \in \Sigma$ ,
- $L(e_1; e_2) = L(e_1) \circ L(e_2)$  for regular expressions  $e_1$  and  $e_2$ ,

- $L(e_1|e_2) = L(e_1) \cup L(e_2)$  for regular expressions  $e_1$  and  $e_2$ ,
- $L(e^*) = L(e)^*$  for a regular expression  $e$ .

Regular languages are recognised by finite state automata.

## 2.2 Finite State Automata

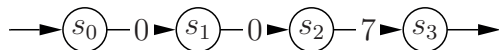
A finite state automaton is a computational model that recognises regular languages. It comprises a finite set of states, an input alphabet, and transitions between its states depending on symbols of its input alphabet. Moreover, a finite state automaton designates two special kinds of states: a start state where it begins its execution, and a set of final states which indicate the proper end of an execution. The following definitions are taken from [HMU06] and adapted when necessary, to the use in this thesis.

### Definition 8. Finite state automaton

A finite state automata is a system  $A = (S, I, d, s_0, F)$  where

- $S$  is a finite set of states,
- $I$  is a finite set of input symbols,
- $d$  is a state transition relation,  $d \subseteq S \times I \times S$ ,
- $s_0 \in S$  is the start state, and
- $F \subseteq S$  is a set of final (or accepting) states.

Finite state automata are often represented by transition diagrams. Thereby, a state is depicted by a node and a transition between two states is depicted by an arc labelled with the respective symbol. To distinguish the start state it is equipped with an incoming arrow, which is not originated in another node. Analogously the final states are equipped with outgoing arrows. In the following a transition diagram for the finite state automaton  $A = (\{s_0, s_1, s_2, s_3\}, \{0, 7\}, \{(s_0, 0, s_1), (s_1, 0, s_2), (s_2, 7, s_3)\}, s_0, \{s_3\})$  is depicted.





## Execution and language recognition

Given an input word, the automaton processes it symbol by symbol from left to right. Beginning from its start state it makes a transition for each symbol from its current state to a follower state. These transitions continue until the entire input word is processed or the automaton gets stuck, i.e. there is no transition from the current state labelled with the current symbol. When the entire input word is processed and the automaton has reached one of its final states, the word is considered to be recognised by the automaton.

The set of all words a given automaton recognises is the *language* of the automaton. In order to define the language formally, the following definition extends the state transition relation  $d$ , processing symbols, to a relation  $d^*$ , processing entire words.

### Definition 9. Extended state transition $d^*$

The extended state transition  $d^*$  is recursively defined by

- (i)  $d^*(s, \lambda) = \{s\}$ ,
- (ii)  $d^*(s, wx) = \bigcup_{\bar{s} \in d^*(s, w)} d(\bar{s}, x)$ .

### Definition 10. Language of a finite state automaton

The language recognised by a finite state automaton  $A = (S, I, d, s_0, F)$  is given by

$$L(A) = \{w \in I^* \mid d^*(s_0, w) \cap F \neq \emptyset\}.$$

## The language of a finite state automaton is regular

Finite state automata recognise regular languages and, conversely, regular languages are recognised by finite state automata. For a proof see, e.g., [Ric08].

## Deterministic finite state automata

The finite state automata considered so far are non deterministic, i.e., for every state and every input symbol  $x$  they may provide a choice of possible transitions to follower states, including that no transition is provided. Deterministic finite state automata provide for each state and each symbol exactly one transition. Formally, deterministic automata are defined as in Definition 8, except for the state transition relation. This now can be regarded as a function  $d : S \times I \rightarrow S$ .

## 2.2.1 Composition of Automata

Finite state automata can be composed sequentially, in choice, and iteratively like regular expressions. The following section defines this compositions formally as given in [KK14]. Also a further composition is recalled, the product automaton, which composes two automata in parallel, such that the resulting automaton executes both input automata simultaneously. For the following definitions let  $A_1 = (S_1, I_1, d_1, s_{0_1}, F_1)$ ,  $A_2 = (S_2, I_2, d_2, s_{0_2}, F_2)$ , and  $A = (S, I, d, s_0, F)$  be finite state automata with disjoint state sets.

The sequential composition of two automata  $A_1$  and  $A_2$  combines them in such a way that  $A_1$  is executed first and subsequently  $A_2$ .

### Definition 11. Sequential composition

$A_1 \circ A_2 = (S, I, d, s_{0_1}, F)$  with

- $S = S_1 \cup S_2$ ,
- $I = I_1 \cup I_2$  (please note that for deterministic automata  $I_1$  and  $I_2$  have to be the same),
- $d = d_1 \cup d_2 \cup \{(s, x, s') \mid (s_{0_2}, x, s') \in d_2, s \in F_1\}$ ,
- $F = \begin{cases} F_1 \cup F_2 & \text{if } s_{0_2} \in F_2 \\ F_2 & \text{otherwise.} \end{cases}$

$A_1 \circ A_2$  recognises  $L(A_1) \circ L(A_2)$ .

The choice composition of two automata  $A_1$  and  $A_2$  results in an automaton which executes either  $A_1$  or  $A_2$ .

### Definition 12. Choice composition

$A_1 \cup A_2 = (S, I, d, s_0, F)$  with

- $S = (S_1 \cup S_2) \cup \{s_0\}$  with  $s_0 \notin S_1 \cup S_2$ ,
- $I = I_1 \cup I_2$  (please note that for deterministic automata  $I_1$  and  $I_2$  have to be the same),
- $d = d_1 \cup d_2 \cup \{(s_0, x, s') \mid (s_{0_1}, x, s') \in d_1\} \cup \{(s_0, x, s') \mid (s_{0_2}, x, s') \in d_2\}$ ,
- $F = \begin{cases} F_1 \cup F_2 \cup \{s_0\} & \text{if } s_{0_i} \in F_i, i \in \{1, 2\} \\ F_1 \cup F_2 & \text{otherwise.} \end{cases}$

$A_1 \cup A_2$  recognises  $L(A_1) \cup L(A_2)$ .

The iteration of an automaton  $A$  results in an automaton, which is able to execute the input automaton arbitrarily often.

**Definition 13. Iteration**

$A^* = (S_*, I, d_*, s_{0_*}, F_*)$  with

- $S_* = S \cup \{s_{0_*}\}$  with  $s_{0_*} \notin S$ ,
- $d_* = d \cup \{(s, x, s') \mid (s_0, x, s') \in d, s \in F \cup \{s_{0_*}\}\}$ ,
- $F = F_A \cup \{s_{0_*}\}$ .

$A^*$  recognises  $L(A^*)$ .

The product automaton is a composition of two deterministic automata, executing both input automata simultaneously.

**Definition 14. Product automaton**

$A_1 \times A_2 = (S_1 \times S_2, I, d, (s_{0_1}, s_{0_2}), F_1 \times F_2)$  with

$d((s_1, s_2), x) = (d_1(s_1, x), d_2(s_2, x))$  for all  $(s_1, s_2) \in S_1 \times S_2$  and  $x \in I$ .

$A_1 \times A_2$  recognises  $L(A_1) \cap L(A_2)$ .

## 2.2.2 Transform Regular Expressions to Finite State Automata

As said above, regular expressions and finite state automata are equivalent, i.e. languages accepted by finite state automata are exactly those, which can be specified by regular expressions. So every regular expression can be transformed to a finite state automaton. Although there are many ways to translate regular expressions to finite state automata we prefer a special kind, since we need automata without  $\lambda$  transitions. Also we presuppose that the automata may not be deterministic. The following definitions are taken from [KK14], except for item 1.. There we added a final state, in order to ensure that the compositions we define in the course of the thesis do not get stuck.

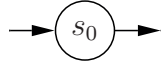
**Definition 15. Regular expressions to finite state automata**

Let  $\Sigma$  be an alphabet and  $\sigma \in \Sigma$ . Moreover, let  $C_1, C_2$ , and  $C$  be regular expressions over  $\Sigma$ . Finite state automata from regular expressions are recursively defined by

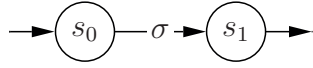
- $A(\emptyset) = (\{s_0, s_1\}, \emptyset, \emptyset, s_0, \{s_1\})$ ,



- $A(\lambda) = (\{s_0\}, \emptyset, \emptyset, s_0, \{s_0\})$ ,



- $A(\sigma) = (\{s_0, s_1\}, \{\sigma\}, \{(s_0, \sigma, s_1)\}, s_0, \{s_1\})$ ,



- $A(C_1; C_2) = A(C_1) \circ A(C_2)$ ,
- $A(C_1|C_2) = A(C_1) \cup A(C_2)$ ,
- $A(C^*) = A(C)^*$ .

## 2.3 Graphs and Rule-based Graph Transformation

This section introduces the main concepts of rule based graph transformation as we use them in this thesis. We employ the double-push out graph transformation approach as addressed in, e.g., [EPS73, Ehr79, CMR<sup>+</sup>96]. For further information about graphs and different graph types see [Roz97].

### 2.3.1 Graphs and Operations Regarding Graphs

Basically graphs consists of nodes and edges between these nodes. One may interpret the nodes as objects and the edges as relations between these objects. Edges can be directed and undirected. Usually one edge connects two nodes. But there are also graphs with edges connecting more than two nodes, called hypergraphs with hyperedges. Nodes and edges can be labelled in order to associate them with further information. There is a variety of graphs in literature with different ways to model the components, each variant suited for particular applications. We use directed, edge-labelled, and

multiple graphs with binary edges, i.e., graphs with directed edges, where every edge connects two nodes and nodes can be connected by multiple edges. Using directed edges one can also model undirected edges by putting two opposed edges between the respective nodes. The following definitions are taken from [KKR08] and [KKK06].

**Definition 16. Graph**

Let  $\Sigma$  be a set of labels. A *graph* over  $\Sigma$  is a system  $G = (V, E, s, t, l)$  where  $V$  is a finite set of *nodes*,  $E$  is a finite set of *edges*,  $s, t: E \rightarrow V$  are mappings assigning a *source*  $s(e)$  and a *target*  $t(e)$  to every edge in  $E$ , and  $l: E \rightarrow \Sigma$  is a mapping assigning a label to every edge in  $E$ . An edge  $e$  with  $s(e) = t(e)$  is also called a *loop*.

The components  $V$ ,  $E$ ,  $s$ ,  $t$ , and  $l$  of  $G$  are also denoted by  $V_G$ ,  $E_G$ ,  $s_G$ ,  $t_G$ , and  $l_G$ , respectively. The set of all graphs over  $\Sigma$  is denoted by  $\mathcal{G}_\Sigma$ . We reserve a specific label  $*$  which is omitted in drawings of graphs. In this way, graphs where all edges are labelled with  $*$  may be seen as *unlabelled graphs*.

A subgraph is a part of a given graph, which is a graph by itself.

**Definition 17. Subgraph**

A graph  $G \in \mathcal{G}_\Sigma$  is a *subgraph* of a graph  $H \in \mathcal{G}_\Sigma$ , denoted by  $G \subseteq H$ , if  $V_G \subseteq V_H$ ,  $E_G \subseteq E_H$ ,  $s_G(e) = s_H(e)$ ,  $t_G(e) = t_H(e)$ , and  $l_G(e) = l_H(e)$  for all  $e \in E_G$ .

**Alter a graph by removing and adding elements**

A given graph can be modified by adding or removing some nodes or edges. Regarding the deletion of elements from a graph some conditions have to be taken into account in order to ensure that the resulting structure again is a graph. Formally, let  $G = (V, E, s, t, l)$  be a graph and  $X = (V_X, E_X) \subseteq (V, E)$  be a pair of sets of nodes and edges. Then  $G - X = (V - V_X, E - E_X, s', t', l')$  with  $s'(e) = s(e)$ ,  $t'(e) = t(e)$ , and  $l'(e) = l(e)$  for all  $e \in E - E_X$  is a subgraph of  $G$  if and only if there is no  $e \in E - E_X$  with  $s(e) \in V_X$  or  $t(e) \in V_X$ . This condition is called *contact condition* of  $X$  in  $G$ .

Instead of removing nodes and edges, one may add some nodes and edges to extend a graph such that the given graph is a subgraph of the extension. The addition of nodes needs no further activities, whereas the addition of edges requires the specification of their labels, sources, and targets, where the latter two may be given in the considered graph or as new nodes. Formally, let  $G = (V, E, s, t, l)$  be a graph and  $(V', E', s', t', l')$  be a structure consisting of

two sets  $V'$  and  $E'$  and three mappings  $s' : E' \rightarrow V \uplus V'$ ,  $t' : E' \rightarrow V \uplus V'$ , and  $l' : E' \rightarrow \Sigma$  (where  $\uplus$  denotes the disjoint union of sets). Then  $H = G + (V', E', s', t', l') = (V \uplus V', E \uplus E', s'', t'', l'')$  is a graph with  $G \subseteq H$  (which establishes the definition of the three mappings  $s'', t'', l''$  on  $E$ ) and  $s''(e') = s'(e')$ ,  $t''(e') = t'(e')$ , and  $l''(e') = l'(e')$  for all  $e' \in E'$ .

### Disjoint union of graphs

If  $G$  is extended by a full graph  $G' = (V', E', s', t', l')$  the graph  $G + G'$  is the *disjoint union* of  $G$  and  $G'$ . Note that in this case  $s'$  and  $t'$  map  $E'$  to  $V'$  rather than  $V \uplus V'$ , but  $V'$  is included in  $V \uplus V'$  such that the extension works. The disjoint union of graphs puts graphs together without any interconnection. If graphs are disjoint, their disjoint union is just the union. If they are not disjoint, the shared nodes and edges must be made different from each other. Because this part of the construction is not made explicit, the disjoint union is only unique up to isomorphism, i.e. up to naming.

### Graph morphisms

Graph morphisms are mappings between graphs which are structure preserving, i.e. source and target of an edge are always mapped onto the source and the target of the image of the edge.

#### Definition 18. Graph morphism and match

For graphs  $G, H \in \mathcal{G}_\Sigma$ , a *graph morphism*  $g : G \rightarrow H$  is a pair of mappings  $g_V : V_G \rightarrow V_H$  and  $g_E : E_G \rightarrow E_H$  that are structure-preserving, i.e.,  $g_V(s_G(e)) = s_H(g_E(e))$ ,  $g_V(t_G(e)) = t_H(g_E(e))$ , and  $l_H(g_E(e)) = l_G(e)$  for all  $e \in E_G$ . A graph morphism  $g$  is injective if  $g_V$  and  $g_E$  are injective.

For a graph morphism  $g : G \rightarrow H$ , the image of  $G$  in  $H$  is called a *match* of  $G$  in  $H$ , i.e., the match of  $G$  with respect to the morphism  $g$  is the subgraph  $g(G) \subseteq H$ . If  $g$  is injective, the match  $g(G)$  is also called *injective*.

### 2.3.2 Rule-based Graph Transformation

Graphs can be transformed by rules. Very roughly speaking a rule partially consists of two graphs called left-hand side and right-hand side. When applying a rule to a graph  $G$  a match of its left-hand side in  $G$  is found and replaced by the right-hand side. Transforming graphs with rules one is able to model processes on graphs.

A rule changes a graph by deleting respectively adding some elements. In order to ensure that the result of a rule application again is a graph, i.e. there are no dangling edges after the rule application, a rule also consists of a gluing graph. The gluing graph is a subgraph of the left- as well as of the right-hand side of the rule and it is preserved when applying the rule. In order to avoid dangling edges a rule only can be applied if all the nodes of the match of the left-hand side adjacent to edges of the surrounding graph are also part the gluing graph.

**Definition 19. Graph transformation rule**

A rule  $r = (L \supseteq K \subseteq R)$  consists of three graphs  $L, K, R \in \mathcal{G}_\Sigma$  such that  $K$  is a subgraph of  $L$  and  $R$ . The components  $L, K$ , and  $R$  of  $r$  are called *left-hand side*, *gluing graph*, and *right-hand side*, respectively.

The class of all rules is denoted by  $\mathcal{R}$ .

The following picture shows a rule which deletes an edge between two nodes and adds an edge labelled with  $x$ .

$$r: \bullet \rightarrow \bullet \supseteq \bullet \quad \bullet \subseteq \bullet - x \rightarrow \bullet$$

**Application of a graph transformation rule**

The application of a graph transformation rule to a graph  $G$  consists of replacing an injective match of the left-hand side in  $G$  by the right-hand side in such a way that the match of the gluing graph is kept. Hence, the application of  $r = (L \supseteq K \subseteq R)$  to a graph  $G = (V, E, s, t, l)$  consists of the following three steps.

1. A graph morphism  $g : L \rightarrow G$  is chosen to establish a match of  $l$  in  $G$  subject to the following two application conditions:
  - a) Contact condition of  $g(L) - g(K) = (g(V_L) - g(V_K), g(E_L), g(E_K))$  in  $G$ ; and
  - b) Identification condition. If two nodes or edges in  $L$  are identified in the match of  $L$  they must be in  $K$ .
2. Now the match of  $L$  up to  $g(K)$  is removed from  $G$ , resulting in a new intermediate graph  $Z = G - (g(L) - g(K))$ .
3. Afterwards the right-hand side  $R$  is added to  $Z$  by gluing  $Z$  with  $R$  in  $g(K)$  yielding the graph  $H = Z + (R - K, g)$  where  $(R - K, g) = (V_R - V_K, E_R - E_K, s', t', l')$  with

$$\begin{aligned}
s'(e') &= s_R(e') \text{ if } s_R(e') \in V_R - V_K \text{ and} \\
s'(e') &= g(s_R(e')) \text{ otherwise,} \\
t'(e') &= t_R(e') \text{ if } t_R(e') \in V_R - V_K \text{ and} \\
t'(e') &= g(t_R(e')) \text{ otherwise, and} \\
l'(e') &= l_R(e') \text{ for all } e' \in E_R - E_K.
\end{aligned}$$

The application of a rule  $r$  to a graph  $G$  is denoted by  $G \xRightarrow[r]{\Rightarrow} H$ , where  $H$  is the graph resulting from the application of  $r$  to  $G$ . The subscript  $r$  may be omitted if it is clear from the context. An arbitrary rule application from a ruleset  $P$  is denoted by  $G \xRightarrow[P]{\Rightarrow} H$ .

### Derivation and application sequence

A rule application is called a *direct derivation*. The semantics of a rule then can be defined by the set of all direct derivations.

#### Definition 20. Semantics of a rule

Let  $r \in \mathcal{R}$  be a rule. The semantics of  $r$  is defined by

$$SEM(r) = \{G \xRightarrow[r]{\Rightarrow} H \mid G, H \in \mathcal{G}\}.$$

Alternatively one may use only the graph pairs as semantics

$$SEM(r) = \{(G, H) \mid G \xRightarrow[r]{\Rightarrow} H, G, H \in \mathcal{G}\}.$$

The sequential application of rules is called *derivation* and formally defined as sequential composition of direct derivations.

#### Definition 21. Derivation

Let  $P \subseteq \mathcal{R}$  be a set of rules. The sequential composition of direct derivations is given by  $d = G_0 \xRightarrow[r_1]{\Rightarrow} G_1 \xRightarrow[r_2]{\Rightarrow} \dots \xRightarrow[r_n]{\Rightarrow} G_n$  with  $n \in \mathbb{N}$ ,  $r_i \in P$  and called *derivation*

from  $G_0$  to  $G_n$ . The derivation from  $G_0$  to  $G_n$  can also be denoted by  $G_0 \xRightarrow[P]{\xRightarrow{n}} G_n$

where  $\{r_1, \dots, r_n\} \subseteq P$ , or by  $G_0 \xRightarrow[P]{*} G_n$  if the length  $n$  is negligible, or just by  $G_0 \xRightarrow{*} G_n$  if  $P$  is clear from the context.

The string  $r_1 \dots r_n$  is called *application sequence* of the derivation  $d$ .

The set of all derivation over a given ruleset  $P$  is denoted by  $der(P)$  and defined as follows.

#### Definition 22. Set of all derivations over a ruleset

Let  $P \subseteq \mathcal{R}$  be a set of rules. Then the set of all finite and infinite derivations over  $P$  is defined by

$$\begin{aligned}
der(P) &= \{G_0 \xRightarrow[r_1]{\Rightarrow} G_1 \xRightarrow[r_2]{\Rightarrow} \dots \xRightarrow[r_n]{\Rightarrow} G_n \mid r_i \in P, G_i \in \mathcal{G}, i \in [n], n \in \mathbb{N}\} \cup \{G_0 \xRightarrow[r_1]{\Rightarrow} \\
&G_1 \xRightarrow[r_2]{\Rightarrow} \dots \mid r_i \in P, G_i \in \mathcal{G}, i \in \mathbb{N}\}.
\end{aligned}$$



### 2.3.3 Graph Transformation Units

Graph transformation units allow to transform graphs in a directed way. A graph transformation unit can be seen as an entity which is able to perform a specific task. Once created it can be invoked whenever it is needed. Furthermore, structured transformation units could be used to modularise a task. For every subtask a unit is constructed. These units then are imported and invoked by a superior unit which models the whole task accomplishment. Such a superior unit is called structured transformation unit. Structured units make the understanding and usage more intuitive and easy.

Formally, transformation units base on an arbitrary graph transformation approach. Since we use the double pushout approach we refrain from defining its general components and instead directly introduce the components as we use them in the thesis. A simple transformation unit comprises two *graph class expressions* specifying permitted *initial* graphs, i.e., where to start the transformation process, and analogously describing desired *terminal* graphs. In order to transform graphs a simple transformation unit provides a set of rules and a *control condition* helping to reduce the nondeterminism of rule application. Graphs, rules, and rule application were already addressed above. In the following we consider graph class expressions and control conditions in more detail. The following definitions are taken from [Kus00, KKR08].

#### Graph class expressions

A *graph class expression* may be any syntactic entity  $X$  that specifies a class of graphs  $SEM(X) \subseteq \mathcal{G}_\Sigma$ . A typical example is the subset  $\Delta \subseteq \Sigma$  with  $SEM(\Delta) = \mathcal{G}_\Delta \subseteq \mathcal{G}_\Sigma$ , i.e.  $SEM(\Delta)$  comprises all graphs, labeled with symbols of  $\Delta$ . Forbidden structures are also frequently used. Let  $F$  be a graph, then  $SEM(forbidden(F))$  contains all graphs  $G$  such that there is no graph morphism  $f : F \rightarrow G$ . The class of possible graph class expressions for transformation units, as we use them in the thesis, is denoted by  $\mathcal{X}$ .

#### Control conditions

A control condition may be any syntactic expression which reduces the non-determinism of possible rule applications. A control condition may give some sort of preferences up to explicitly describe permitted rule application sequences. Often, the semantics of a control condition is given by a pair of graphs, i.e. given a control condition  $C$ ,  $C$  specifies binary relation  $SEM(C) \subseteq \mathcal{G} \times \mathcal{G}$ . It is also possible, as done in this thesis, to specify the semantics of a control condition as set of permitted derivations, i.e.

$SEM(C) \subseteq der(P)$  (respectively  $der(P)$  resctricted to finite derivations). A variety of expressions can be applied as control conditions. In the following some examples are presented. For all examples let  $P \subseteq \mathcal{R}$  be a set of rules.

**Priorities on Rules** A priority control condition is a irreflexive partial order on a set of rules. The intuitive meaning is that rules of a lower priority only can be applied when no rules of higher priorities are applicable. Formally given a set of rules  $P$ , a priority is a pair  $C = (P, <)$  where  $<$  is an irreflexive partial order on  $P$ . For  $r \in P$ , let  $HP_C(r) = \{r' \in P \mid r < r'\}$  denote the set of rules with higher priority in  $C$ . Then  $(G, H) \in SEM(C)$  if there are  $G_0, \dots, G_n \in \mathcal{G}$  such that

- $G_0 = G$  and  $G_n = H$ ,
- for  $i = 1, \dots, n$ ,  $(G_{i-1}, G_i) \in SEM(r_i)$  for some  $r_i \in P$ , and for all  $r \in HP_C(r_i)$  there is no  $G \in \mathcal{G}$  with  $(G_{i-1}, G) \in SEM(r)$ .

**Regular Expressions** Regular expressions over rules are often used as control conditions. They describe permitted rule application sequences.

$SEM(e) = \{(G, H) \mid G \xrightarrow{*} H \in der(P) \text{ and the language of } e, L(e), \text{ is application sequence of } G \xrightarrow{*} H\}$

**As-Long-As-Possible** Given a set of rules  $P$  the control condition as-long-as-possible requires that all the rules must be applied as long as possible, i.e. it allows all derivations  $G \xRightarrow{P} H$  such that no rule of  $P$  is applicable to  $H$ .

$SEM(as-long-as-possible) = \{(G, H) \mid G \xrightarrow{*} H \in der(P), \nexists H \xrightarrow[r]{} H', \forall r \in P\}$

The class of possible control conditions for transformation units, as we use them in the thesis, is denoted by  $\mathcal{C}$ . Other control conditions for transformation units may be found in [Kus98].

Formally, a simple transformation unit is defined by

**Definition 23. Simple graph transformation unit**

A simple graph transformation unit is a system  $tu = (I, P, C, T)$ , where

- $I \in \mathcal{X}$  and  $T \in \mathcal{X}$  are graph class expressions to specify the *initial* and the *terminal* graphs respectively,
- $P \subseteq \mathcal{R}$  is a set of rules, and
- $C \in \mathcal{C}$  is a control condition.

The semantics of a simple transformation unit is given by a set of graph pairs, each containing the initial and the terminal graph of a transformation process permitted by the control condition.

**Definition 24. Semantics of a simple transformation unit**

Let  $tu = (I, P, C, T)$  be a simple transformation unit. The semantic relation is given by

$$SEM(tu) = \{(G, H) \mid G \xrightarrow[P]{*} H, G \in SEM(I), H \in SEM(T), (G, H) \in SEM(C)\}.$$

For complex tasks the rule set of a transformation unit can be very large, also it is very convenient to be able to reuse transformation units for already solved problems. For this reasons modularisation is worthwhile. With modularisation large rule sets can be structured, so the interaction of the different rules is better understandable. Solutions of old problems can be reused for solving new ones. In order to obtain modularisation, simple transformations units can, besides rules, be equipped with other transformations units, called imported units. Since transformation units provide a binary relation as semantics, imported units can be used like rules. The resulting unit than is called *structured transformation unit* or *transformation unit with import*.

**Definition 25. Transformation unit with import**

A transformation unit with import is a system  $tu = (I, P, U, C, T)$ , where  $(I, P, C, T)$  is a simple transformation unit, and the component  $U$  is a set of imported transformation units.

In order to avoid that a transformation unit is allowed to directly or indirectly import itself, we assume an acyclic imported structure. To achieve this transformation units are assigned with an import level, where level 0 stands for 'no units are imported' ( $U = \emptyset$ ), and each unit can only import units from a lower level.

The semantics of a structured transformation unit not only has to consider the rules but also the imported units. Rule applications alternate with calls of imported units. Since, like rules, the semantics of a unit is a binary relation on graphs an imported transformation unit can be handled like a rule. This leads to a so called *interleaving semantics*.

**Definition 26. Interleaving semantics**

Let  $tu = (I, P, U, C, T)$  be a structured transformation unit. Its interleaving semantic relation is defined by

$$INTER_{SEM}(tu) = \{(G, G') \mid G \in SEM(I), G' \in SEM(T), \text{ and there is a}$$

sequence  $G_0, \dots, G_n$  with  $G = G_0, G_n = G'$ , and, for  $i = 1, \dots, n$ ,  $G_{i-1} \xrightarrow{r} G_i$  for some  $r \in P$  or  $(G_{i-1}, G_i) \in SEM(u)$  for some  $u \in U$ . Moreover,  $(G, G')$  must be accepted by the control condition  $C$ .

### 2.3.4 Parallel Graph Transformation Rules

Until now the provided rules of a transformation unit have been applied sequentially. With *parallel rules* as, e.g., discussed in [Kre78, Roz97] one is able to apply two (or more) rules to a graph simultaneously. The following definition presents parallel rules as they are defined in [KK07].

**Definition 27. Parallel rule**

Let  $r_1 = (L_1 \supseteq K_1 \subseteq R_1)$  and  $r_2 = (L_2 \supseteq K_2 \subseteq R_2)$  be two rules. Then  $r_1 + r_2 = (L_1 + L_2 \supseteq K_1 + K_2 \subseteq R_1 + R_2)$  is called the *parallel rule* of  $r_1$  and  $r_2$ .

In order to denote the set of all parallel rules over a given rule set, we employ the definition given in [Hab04], but with modified notation.

**Definition 28. Set of all parallel rules  $\mathcal{e}$  over a set of rules**

For a rule set  $\mathcal{R}$  the set of all parallel rules,  $\mathcal{R}_*$ , is inductively defined by:

1.  $\mathcal{R} \subseteq \mathcal{R}_*$  and
2. for  $r_1, r_2 \in \mathcal{R}_*$  also  $r_1 + r_2 \in \mathcal{R}_*$

For more information about parallelism in graph transformation see e.g. [EEPT06, EKMR99].

## 2.4 Petri Nets

In this section Petri nets and some of their properties used in the thesis are introduced. The definitions are taken from [Mur89] and [Bau96]. We have adapted some definitions to meet our requirements. For example we do not need the distinction between a Petri net graph (Petri net without initial marking) and a Petri net, so we have modified the original definitions regarding Petri net graphs to definitions for Petri nets.

Petri nets are a means to model dynamic and concurrent systems. In the thesis we employ them to represent the deletion and addition of graph elements described by a control condition. Petri nets are graphs which have

two types of nodes, places and transitions. Roughly speaking, places model states and transitions model events altering states. Places and transitions are linked by weighted arcs. Every place holds a number of tokens represented by a natural number. The number of tokens on places linked to a transition determines if the transition can happen, respectively *fire* in the notation of Petri nets. The initial assignment of the places with tokens is called *initial marking* of the Petri net.

## Basic Definitions

### Definition 29. Petri net

A Petri net is a 5-tuple,  $PN = (P, T, F, W, M_0)$  where

- $P$  is a finite set of places,
- $T$  is a finite set of transitions,
- $F \subseteq (P \times T) \cup (T \times P)$  is a set of arcs called flow relation,
- $W : F \rightarrow \mathbb{N}^+$  is a weight function, and
- $M_0 : P \rightarrow \mathbb{N}$  is the initial marking.

Moreover, it holds that  $P \cap T = \emptyset$  and  $P \cup T \neq \emptyset$ .

The dynamics of a Petri net is modelled by *firing* its transitions. In order to define the firing of a transition formally, two notions are stated: the set of all *input places* regarding a transition  $t$ ,  $\bullet t = \{p \mid (p, t) \in F\}$ , and analogously the set of all *output places* regarding a transition  $t$ ,  $t\bullet = \{p \mid (t, p) \in F\}$ . A transition is able to fire if the number of tokens on its input places is at least as large as the weight of the respective arc. If a transition fires two things happen: the number of tokens on the input places of the transition is decreased by the weight of the respective arcs, and the number of tokens on the output places is increased by the weight of the respective arcs. In case the Petri net is *bounded*, i.e. the amount of tokens a place can hold is limited, it is also necessary that the increased number of tokens does not exceed the limitations for the respective places.

### Definition 30. Marking, fire, and firing sequence

Let  $PN = (P, T, F, W, M_0)$  be a Petri net.

1. Any mapping  $M : P \rightarrow \mathbb{N}$  is called a *marking* of  $PN$ .
2. A transition  $t \in T$  is *enabled* at marking  $M$ , denoted by  $M[t >$ , if  $W(p, t) \leq M(p) \forall p \in \bullet t$ .

3. Then  $t$  may *fire* reaching a marking  $M'$  defined by

$$M'(p) = \begin{cases} M(P) - W(p, t) & \text{if } p \in \bullet t \setminus t \bullet, \\ M(p) + W(t, p) & \text{if } p \in t \bullet \setminus \bullet t, \\ M(P) - W(p, t) + W(t, p) & \text{if } p \in t \bullet \cap \bullet t, \\ M(p) & \text{otherwise.} \end{cases}$$

The firing of a transition  $t$  from marking  $M$  reaching marking  $M'$  is denoted by  $M [t > M'$ .

4. A transition sequence  $t_1 \cdots t_n \in T^*$  is a *firing sequence* starting from marking  $M$ , denoted by  $M [t_1 \cdots t_n > M'$ , if there are markings  $M_0, M_1, \dots, M_n$  such that  $M = M_0$ ,  $M_{i-1} [t_i > M_i$  for all  $i \in [n]$ , and  $M_n = M'$ .

Every firing sequence could be mapped to a vector called *Parikh-vector* which states for every transition the frequency of its occurrence in the firing sequence.

**Definition 31. Parikh-vector**

Let  $PN = (P, T, F, W, M_0)$  be a Petri net and  $w$  be a firing sequence. The vector  $\bar{w} = \begin{pmatrix} \#(t_1, w) \\ \vdots \\ \#(t_n, w) \end{pmatrix}$  with  $t_1, \dots, t_n \in T, n = \#T$ , and  $\#(t, w)$  denotes the occurrence of  $t$  in the firing sequence  $w$ , is called Parikh-vector.

The Parikh-vector abstracts from the actual sequential order of the transitions, so one Parikh-vector represents a whole set of firing sequences.

Another means of description, regarding the behaviour of a Petri net, is the incidence matrix. The incidence matrix of a Petri net represents for every transition (columns) the modification at the places (rows) after firing the respective transition.

**Definition 32. Incidence matrix**

Let  $PN = (P, T, F, W, M_0)$  be a Petri net. The incidence matrix  $A$  of the Petri net is defined by

$$A_{ij} := \begin{cases} W(t_j, s_i) & \text{if } (t_j, s_i) \in F \setminus F^{-1}, \\ -W(s_i, t_j) & \text{if } (s_i, t_j) \in F \setminus F^{-1}, \\ W(t_j, s_i) - W(s_i, t_j) & \text{if } (t_j, s_i) \in F \cap F^{-1}, \\ 0 & \text{otherwise,} \end{cases}$$

for the indexes  $i \in [\#P], j \in [\#T]$ .

A Petri net is *pure* if it has no self-loops, i.e. there is no place  $\in P$  which has in- and outgoing arcs from and to the same transition. If a Petri net is pure the underlying graph of the Petri net is uniquely given, up to naming, by the incidence matrix.

## Properties

### Definition 33. Partial repetitiveness

A Petri net is *partially repetitive* if there exists an infinite firing sequence  $w = (t_i)_{i \in \mathbb{N}}$  starting from the initial marking  $M_0$ .

Partial repetitivity can be characterised with the help of the incidence matrix.

**Theorem 34.** Let  $PN = (P, T, F, W, M_0)$  be a Petri net and  $A$  its incidence matrix.  $PN$  is partially repetitive if and only if there exists a  $|T|$ -vector  $x: T \rightarrow \mathbb{N}$  such that  $A \cdot x \geq 0$  and  $x \neq 0$ .

# Chapter 3

## Parallel Control Conditions

Modelling processes with graph transformation, the ability to express parallelism is highly beneficial for mainly two reasons. Firstly, parallelism is an inherent property of the real world and the adequate modelling of real world processes needs to express parallelism. Secondly, parallel execution of processes saves time. This chapter introduces *parallel expressions*, which are able to directly describe parallel graph transformation processes. Parallel expressions implement three kinds of parallelism by augmenting regular expressions by respectively one parallel composition operator.

Since the notion of parallelism is very general and highly related to the field of application we first have to specify the terms we use throughout the thesis regarding parallelism in graph transformation. Thereby we distinguish three forms of parallelism, namely weak, proactive, and synchronous parallelism. These forms of parallelism are then implemented by three kinds of parallel expressions (weak, proactive, and synchronous expressions) providing respective parallel composition operators. This chapter focuses on weak and synchronous expressions. Proactive expressions are addressed in chapter 7, when considering stepwise controls. Paying attention to weak and synchronous expressions the chapter examines their composition operators on the syntactic level and states some algebraic laws. In order to employ parallel expressions as control condition, providing permitted derivations, two approaches are presented. The first approach is based on the language of weak and synchronous expressions. The second approach employs a canonical form for synchronous and weak parallel expressions in order to gradually identify which rules have to be applied simultaneously in each derivation step. The chapter concludes with two examples presenting transformation units with synchronous expressions as control conditions.



## 3.1 Parallelism and Graph Transformation

Speaking about parallelism in general often the notions of events and processes are used. Regarding graph transformation we associate these notions to rules and rule sequences. Thus we have to consider parallel application respectively composition of rules and rule sequences.

### Parallel composition of rules

Graph transformation provides the parallel composition of rules, i.e. the simultaneous application of two rules in one derivation step. As a reminder, the parallel composition of rules takes two rules and combines them to one single rule, called parallel rule. Formally, the parallel composition of two rules  $r_1 = (L_1, K_1, R_1)$  and  $r_2 = (L_2, K_2, R_2)$  is given by  $r_1+r_2 = (L_1 + L_2, K_1 + K_2, R_1 + R_2)$  where  $+$  denotes the disjoint union of graphs. A rule application in graph transformation is considered to be atomic, i.e. it can not be interrupted by other rule applications. Provided injective matches the applicability of a parallel rule implies that there are no conflicts between the original rules regarding the current graph.

In [KK07] parallel rules are employed to model several transformation units in a community acting in parallel. Regarding one unit, possible actions of the other units are modelled by so called meta-rules, a set of parallel rules representing what may happen outside the unit. Through meta-rules each unit is enabled to relate its own activities to potential activities of the other units, i.e. the semantics of the control condition of each unit is comprised of applying its own rules composed in parallel to meta-rules. The parallel semantics of the community then consists of all processes fitting to the semantics of all units. Nevertheless in this approach the units themselves are not able to employ parallelism directly in their control conditions. In order to enable a unit to use parallelism its control condition has to be equipped with parallel operators, which is the task of this chapter. More details about parallel rules and further information can be found, e.g., in [Roz97] pages 174 ff..

### Different forms of parallel composition of rule sequences

In order to compose rule sequences in parallel we need to characterise precisely what we intent since there are several possibilities to compose rule sequences, such that rules of the sequences are applied in parallel. In the following we discuss three different forms of parallel composition: weak par-

allel, proactive, and synchronous composition.

*Weak parallel composition* is the weakest form of parallel composition. The rules of the involved sequences can be applied in an arbitrary way. Every possible temporal relation is allowed, as long as the sequential order of rules of the individual sequences is preserved. Weak parallel composition is non-deterministic. There are many possibilities to proceed when composing rule sequences weakly parallel.

*Proactive composition* of sequences is a stronger form of parallel composition. The rules of proactively composed sequences have to be applied simultaneously as soon as possible, i.e. rules of different sequences that can be applied simultaneously have to be applied simultaneously, rules which cannot, can be applied sequentially. So far proactive parallel composition is ambiguous. For illustration, consider the set of all first rules of given sequences. In case these rules could not be applied simultaneously altogether, there may be different partitions into sets of rules which can be applied simultaneously. Regarding these partitions as different equitable possibilities which all may be pursued leads to a non-deterministic proactive composition. Another possibility is to specify which combination to prefer when implementing the proactive composition operator. Proactive composition needs to know the actual graph to which the rule sequences have to be applied to. Without knowing the input graph it is not possible to provide potential rule application sequences resulting from proactively composed rule sequences.

*Synchronous composition* is the strongest form of parallel composition. The respective rules of synchronously composed sequences must be applied simultaneously otherwise the synchronisation fails, i.e. in every derivation step every current rule of each involved sequence has to be applied simultaneously with the others. A rule is not allowed to wait until it is applicable. This description is unambiguous if the involved sequences have the same length. The sequences start and end at the same time and in between all involved rules are applied simultaneously according to the order of their respective sequences. Though if the sequences differ in length there are different possibilities to start the sequences and let them end. One possibility is to force synchronously composed sequences to be of the same length by letting the synchronisation start at the same time and fail if one sequence ends and another not (strong synchronisation). Another possibility is to let the sequences start their synchronous application at the same time but allow them to end differently, i.e. if one sequence ends the others proceed further alone (called begin-synchronisation). For some applications it may also

be useful to let the sequences start differently but require them to end at the same time (end-synchronisation). The last option on this discrimination level is to let them start and end differently as long as they run synchronously once they started (free-synchronisation). This case carries further options of discrimination, which we do not proceed further. The strong-, begin-, and end-synchronous compositions of rule sequences are deterministic. In the course of the thesis we focus on begin-synchronisation and refer to it as synchronisation.

The following definition defines a parallel composition of rule sets which we need later, the product composition of two rule sets. Given two rule sets the product composition comprises all parallel rules build by composing respectively one rule of each set in parallel.

**Definition 35. Parallel product composition of two rule sets**

Let  $P_1, P_2 \in \mathcal{R}$  be two sets of rules. The parallel product composition of  $P_1$  and  $P_2$  is defined by  $P_1 | \times | P_2 = \{r_1 + r_2 \mid r_1 \in P_1, r_2 \in P_2\}$ .

## 3.2 Parallel Expressions

In order to provide control conditions which allow to describe weak parallel, proactive and synchronous composition we introduce weak parallel, proactive, and synchronous expressions subsumed under the notion of *parallel expressions*. Parallel expressions are a generalisation of regular expressions over a rule set  $X$  to expressions over  $X_*$  (the set of all parallel rules over  $X$ ). In order to express parallel composition regular expressions are equipped with respectively one of three additional binary operators, weak parallel composition ( $\mathbb{W}$ ), proactive parallel composition ( $\mathbb{P}$ ), or synchronous composition ( $\mathbb{S}$ ).

**Definition 36. Weak parallel expression**

Let  $X$  be a rule set. Weak parallel expressions over  $X_*$  are recursively defined by

- $\emptyset, \lambda, r \in X_*$  are weak parallel expressions.
- $C_1; C_2, C_1 | C_2, C^*$ , and  $C_1 \mathbb{W} C_2$  are weak parallel expressions if  $C, C_1$ , and  $C_2$  are weak parallel expressions.

**Definition 37. Proactive expression**

Let  $X$  be a rule set. Proactive expressions over  $X_*$  are recursively defined by

- $\emptyset, \lambda, r \in X_*$  are proactive expressions.
- $C_1; C_2, C_1|C_2, C^*$ , and  $C_1\#C_2$  are proactive expressions if  $C, C_1$ , and  $C_2$  are proactive expressions.

**Definition 38. Synchronous expression**

Let  $X$  be a rule set. Synchronous expressions over  $X_*$  are recursively defined by

- $\emptyset, \lambda, r \in X_*$  are synchronous expressions.
- $C_1; C_2, C_1|C_2, C^*$ , and  $C_1\$C_2$  are synchronous expressions if  $C, C_1$ , and  $C_2$  are synchronous expressions.

To avoid brackets we assume that  $*$  binds stronger than  $\$, \forall, \#$  which bind stronger than  $|$ ; which binds stronger than  $|$ .

In the following we pay attention to weak parallel and synchronous expressions, since they differ from proactive expressions and themselves have much in common. Especially, they describe regular languages, as we present in the next chapter. We take up proactive expressions later in 7.2.2 where we address parallel stepwise controls. In the following we introduce the language of weak parallel and synchronous expressions.

### 3.2.1 Language of Weak Parallel and Synchronous Expressions

The language of a weak or synchronous expression is given by the set of rule sequences described by the expression. In order to define the language we first have to define the weak parallel and synchronous composition of words and languages. Afterwards the languages of weak and synchronous expressions are defined and some properties of the weak parallel and synchronous composition of words and languages are introduced.

To keep the following text readable we stick to the notion of parallel expression (respectively composition) although we only refer to the weak and synchronous case for now.

#### Weak and synchronous composition of words and languages

The parallel composition of words reflects the respective composition of rule sequences described in Section 3.1. I.e. the synchronous composition of two words successively composes one symbol from each word in parallel. If

one word comes to an end the remaining part of the other is sequentially composed to the so far constructed parallel composition. The weak parallel composition of words yields all possible compositions of the symbols of the input words, sequentially or in parallel, as long as the sequential order of each input word is preserved.

In the following we define the parallel composition of words. Then on the basis of these definitions we define the parallel composition of languages.

**Definition 39. Weak and synchronous composition of words**

Let  $X$  be an alphabet and  $w, w_1, w_2 \in X_*^*$  be words over  $X_*$ . Moreover, let  $r_1, r_2 \in X_*$ .

a) The weak parallel composition of two words is recursively defined by:

- (i)  $\lambda \mathbb{W} w = \{w\} = w \mathbb{W} \lambda$ ,
- (ii)  $r_1 w_1 \mathbb{W} r_2 w_2 = \{r_1+r_2\}(w_1 \mathbb{W} w_2) \cup \{r_1\}(w_1 \mathbb{W} (r_2 w_2)) \cup \{r_2\}((r_1 w_1) \mathbb{W} w_2)$ .

b) The synchronous composition of two words is recursively defined by:

- (i)  $\lambda \$ w = w = w \$ \lambda$ ,
- (ii)  $(r_1 w_1) \$ (r_2 w_2) = (r_1+r_2)(w_1 \$ w_2)$ .

In order to emphasise the behaviour of the parallel composition each of the following examples demonstrate the evaluation of the parallel composition of two symbols. For each example let  $X$  be the underlying alphabet and  $r_1, r_2 \in X_*$ .

**Example 1.** Weak parallel composition of two symbols

Using definition 39 a), one gets

$$\begin{aligned}
 r_1 \mathbb{W} r_2 &= (r_1 \lambda) \mathbb{W} (r_2 \lambda) \\
 &= \{r_1+r_2\}(\lambda \mathbb{W} \lambda) \cup \{r_1\}(\lambda \mathbb{W} (r_2 \lambda)) \cup \{r_2\}((r_1 \lambda) \mathbb{W} \lambda) \\
 &= \{r_1+r_2\}\{\lambda\} \cup \{r_1\}\{r_2 \lambda\} \cup \{r_2\}\{r_1 \lambda\} \\
 &= \{r_1+r_2\} \cup \{r_1 r_2 \lambda\} \cup \{r_2 r_1 \lambda\} \\
 &= \{r_1+r_2\} \cup \{r_1 r_2\} \cup \{r_2 r_1\} \\
 &= \{r_1+r_2, r_1 r_2, r_2 r_1\}.
 \end{aligned}$$

**Example 2.** Synchronous composition of two symbols

Using definition 39 b), one gets

$$r_1 \$ r_2 = (r_1 \lambda) \$ (r_2 \lambda) = (r_1+r_2)(\lambda \$ \lambda) = (r_1+r_2)\lambda = r_1+r_2.$$

Now we are able to define the parallel composition of two languages, which is given by the parallel composition of the words of the languages.

**Definition 40. Weak and synchronous composition of languages**

Let  $L_1$  and  $L_2$  be two languages over  $X_*$ .

- a) The weak parallel composition of  $L_1$  and  $L_2$  is defined by:

$$L_1 \mathbb{W} L_2 = \bigcup_{w_1 \in L_1, w_2 \in L_2} (w_1 \mathbb{W} w_2).$$

- b) The synchronous composition of  $L_1$  and  $L_2$  is defined by:

$$L_1 \$ L_2 = \{w_1 \$ w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

The language of parallel expressions is analogously defined to the language of regular expressions. The languages for sequentially, alternatively, and iteratively composed expressions are defined as for regular expressions. Additionally, the definition introduces the languages for weakly and synchronously composed expressions. Moreover, the basic alphabet is not  $X$ , but  $X_*$ .

**Definition 41. Language of weak and synchronous expressions**

Let  $C_1, C_2, C$  be weak resp. synchronous expressions over  $X_*$  and  $r \in X_*$ . The language is recursively defined by:

1.  $L(\emptyset) = \emptyset$ ,
2.  $L(\lambda) = \{\lambda\}$ ,
3.  $L(r) = \{r\}$ ,
4.  $L(C_1; C_2) = L(C_1)L(C_2)$ ,
5.  $L(C_1|C_2) = L(C_1) \cup L(C_2)$ ,
6.  $L(C^*) = L(C)^*$ ,
7. a)  $L(C_1 \mathbb{W} C_2) = L(C_1) \mathbb{W} L(C_2)$ ,  
 b)  $L(C_1 \$ C_2) = L(C_1) \$ L(C_2)$ .

The following examples demonstrate the construction of the language of a weak parallel and a synchronous expression.

**Example 3.** Language of a weak parallel expression

Consider the parallel expression  $(r_1; r_2) \mathbb{W} r'_1$ . Its language,  $L((r_1; r_2) \mathbb{W} r'_1)$ , is obtained as follows using definitions 41 and 39 a):

$$\begin{aligned}
& L((r_1; r_2) \mathbb{W} r'_1) \\
&= L((r_1; r_2) \mathbb{W} L(r'_1)) \\
&= (L(r_1)L(r_2)) \mathbb{W} L(r'_1) \\
&= (\{r_1\}\{r_2\}) \mathbb{W} \{r'_1\} \\
&= \{r_1 r_2\} \mathbb{W} \{r'_1\} \\
&= \{r_1+r'_1\}(r_2 \mathbb{W} \lambda) \cup \{r_1\}(r_2 \mathbb{W} r'_1) \cup \{r'_1\}((r_1 r_2) \mathbb{W} \lambda) \\
&= \{r_1+r'_1\}\{r_2\} \cup \{r_1\}(r_2 \mathbb{W} r'_1) \cup \{r'_1\}\{r_1 r_2\} \\
&= \{r_1+r'_1\}\{r_2\} \cup \{r_1\}(\{r_2+r'_1\}(\lambda \mathbb{W} \lambda) \cup \{r_2\}(\lambda \mathbb{W} r'_1) \cup \{r'_1\}(r_2 \mathbb{W} \lambda)) \cup \{r'_1\}\{r_1 r_2\} \\
&= \{r_1+r'_1\}\{r_2\} \cup \{r_1\}(\{r_2+r'_1\}\{\lambda\} \cup \{r_2\}\{r'_1\} \cup \{r'_1\}\{r_2\}) \cup \{r'_1\}\{r_1 r_2\} \\
&= \{(r_1+r'_1)r_2\} \cup \{r_1\}(\{r_2+r'_1\} \cup \{r_2 r'_1\} \cup \{r'_1 r_2\}) \cup \{r'_1 r_1 r_2\} \\
&= \{(r_1+r'_1)r_2\} \cup \{r_1(r_2+r'_1)\} \cup \{r_1 r_2 r'_1\} \cup \{r_1 r'_1 r_2\} \cup \{r'_1 r_1 r_2\} \\
&= \{(r_1+r'_1)r_2, r_1(r_2+r'_1), r_1 r_2 r'_1, r_1 r'_1 r_2, r'_1 r_1 r_2\}
\end{aligned}$$

**Example 4.** Language of a synchronous expression

Consider the parallel expression  $((r_1; r_2; r_3) \$ (r_4; r_5 \$ r_{11})); r_6) \$ (r_7; r_8; r_9)$ . Its language is obtained as follows:

$$\begin{aligned}
& L(((r_1; r_2; r_3) \$ (r_4; r_5 \$ r_{11})); r_6) \$ (r_7; r_8; r_9; r_{10})) \\
&= L(((r_1; r_2; r_3) \$ (r_4; r_5 \$ r_{11})); r_6) \$ L(r_7; r_8; r_9) \quad [\text{def. 41 (7b)}] \\
&= (L((r_1; r_2; r_3) \$ (r_4; r_5 \$ r_{11}))L(r_6)) \$ L(r_7; r_8; r_9) \quad [\text{def. 41 (4)}] \\
&= ((L(r_1; r_2; r_3) \$ L(r_4; r_5 \$ r_{11}))L(r_6)) \$ L(r_7; r_8; r_9) \quad [\text{def. 41 (7b)}] \\
&= ((L(r_1; r_2; r_3) \$ (L(r_4)L(r_5 \$ r_{11})))L(r_6)) \$ L(r_7; r_8; r_9) \quad [\text{def. 41 (4)}] \\
&= ((L(r_1; r_2; r_3) \$ (L(r_4)(L(r_5) \$ L(r_{11}))))L(r_6)) \$ L(r_7; r_8; r_9) \quad [\text{def. 41 (7b)}] \\
&= ((\{r_1 r_2 r_3\} \$ (\{r_4\}(\{r_5\} \$ \{r_{11}\})))\{r_6\}) \$ \{r_7 r_8 r_9\} \quad [\text{def. 41 (4,3)}] \\
&= ((\{r_1 r_2 r_3\} \$ (\{r_4\}(\{r_5 \$ r_{11}\})))\{r_6\}) \$ \{r_7 r_8 r_9\} \quad [\text{def. 40 (b)}] \\
&= ((\{r_1 r_2 r_3\} \$ (\{r_4\}(\{r_5+r_{11}\})))\{r_6\}) \$ \{r_7 r_8 r_9\} \quad [\text{def. 39 (b (i),(ii))}] \\
&= ((\{r_1 r_2 r_3\} \$ \{r_4 r_5+r_{11}\})\{r_6\}) \$ \{r_7 r_8 r_9\} \\
&= (\{(r_1 r_2 r_3) \$ (r_4 r_5+r_{11})\}\{r_6\}) \$ \{r_7 r_8 r_9\} \quad [\text{def. 40 (b)}] \\
&= (\{(r_1+r_4)((r_2 r_3) \$ (r_5+r_{11}))\}\{r_6\}) \$ \{r_7 r_8 r_9\} \quad [\text{def. 39 (b (ii))}] \\
&= (\{(r_1+r_4)(r_2+r_5+r_{11})(r_3 \$ \lambda)\}\{r_6\}) \$ \{r_7 r_8 r_9\} \quad [\text{def. 39 (b (ii))}] \\
&= (\{(r_1+r_4)(r_2+r_5+r_{11})(r_3)\}\{r_6\}) \$ \{r_7 r_8 r_9\} \quad [\text{def. 39 (b (i))}] \\
&= \{r_1+r_4 \ r_2+r_5+r_{11} \ r_3 \ r_6\} \$ \{r_7 \ r_8 \ r_9\} \\
&= \{(r_1+r_4 \ r_2+r_5+r_{11} \ r_3 \ r_6) \$ (r_7 \ r_8 \ r_9)\} \quad [\text{def. 40 (b)}] \\
&= \{(r_1+r_4+r_7) ((r_2+r_5+r_{11} \ r_3 \ r_6) \$ (r_8 \ r_9))\} \quad [\text{def. 39 (b (ii))}] \\
&= \{(r_1+r_4+r_7) (r_2+r_5+r_{11}+r_8) ((r_3 \ r_6) \$ r_9)\} \quad [\text{def. 39 (b (ii))}] \\
&= \{(r_1+r_4+r_7) (r_2+r_5+r_{11}+r_8) (r_3+r_9) (r_6 \$ \lambda)\} \quad [\text{def. 39 (b (ii))}] \\
&= \{(r_1+r_4+r_7) (r_2+r_5+r_{11}+r_8) (r_3+r_9) (r_6)\} \quad [\text{def. 39 (b (i))}]
\end{aligned}$$

### 3.2.2 Properties of Weak and Synchronous Composition

Like other operators on words and languages also the parallel composition operators have some properties. They are commutative, associative, and distributive over  $\cup$ . Moreover it holds, that  $\emptyset$  serves as annihilator and  $\lambda$  as identity. In the following these properties are formulated and proved.

**Proposition 42.** Properties of  $\mathbb{W}$  and  $\$$  for words.

Let  $w_1, w_2, w_3 \in X_*^*$ . The following properties for  $\mathbb{W}$  and  $\$$  hold:

1. Commutativity
  - a)  $w_1 \mathbb{W} w_2 = w_2 \mathbb{W} w_1$ ,
  - b)  $w_1 \$ w_2 = w_2 \$ w_1$ .
2. Associativity
  - a)  $(w_1 \mathbb{W} w_2) \mathbb{W} w_3 = w_1 \mathbb{W} (w_2 \mathbb{W} w_3)$ ,
  - b)  $(w_1 \$ w_2) \$ w_3 = w_1 \$ (w_2 \$ w_3)$ .

**Proposition 43.** Properties of  $\mathbb{W}$  and  $\$$  for languages

Let  $L_1, L_2, L_3, L \subseteq (X_*)^*$ . The following properties for  $\mathbb{W}$  and  $\$$  hold:

1. Commutativity
  - a)  $L_1 \mathbb{W} L_2 = L_2 \mathbb{W} L_1$ ,
  - b)  $L_1 \$ L_2 = L_2 \$ L_1$ .
2. Associativity
  - a)  $(L_1 \mathbb{W} L_2) \mathbb{W} L_3 = L_1 \mathbb{W} (L_2 \mathbb{W} L_3)$ ,
  - b)  $(L_1 \$ L_2) \$ L_3 = L_1 \$ (L_2 \$ L_3)$ .
3. Right distributivity over  $\cup$ 
  - a)  $(L_1 \cup L_2) \mathbb{W} L_3 = (L_1 \mathbb{W} L_3) \cup (L_2 \mathbb{W} L_3)$ ,
  - b)  $(L_1 \cup L_2) \$ L_3 = (L_1 \$ L_3) \cup (L_2 \$ L_3)$ .
4. Left distributivity over  $\cup$ 
  - a)  $L_1 \mathbb{W} (L_2 \cup L_3) = (L_1 \mathbb{W} L_2) \cup (L_1 \mathbb{W} L_3)$ ,
  - b)  $L_1 \$ (L_2 \cup L_3) = (L_1 \$ L_2) \cup (L_1 \$ L_3)$ .
5.  $\emptyset$  is annihilator for  $\mathbb{W}$  and  $\$$ 
  - a)  $L \mathbb{W} \emptyset = \emptyset$ ,
  - b)  $L \$ \emptyset = \emptyset$ .
6.  $\{\lambda\}$  is identity for  $\mathbb{W}$  and  $\$$ 
  - a)  $L \mathbb{W} \{\lambda\} = L$ ,
  - b)  $L \$ \{\lambda\} = L$ .



In order to prove Proposition 42 we need the distributivity of  $\mathbb{W}$  over  $\cup$  for languages stated in Proposition 43. Since the proof for the distributivity of  $\mathbb{W}$  over  $\cup$  for languages needs none of the properties regarding the parallel operators for words we use the needed distributivity already in the following proof but will prove it afterwards. Moreover, we need two additional assertions, since the weak parallel composition of words results in a language. Consider the weak parallel composition of three words  $(w_1 \mathbb{W} w_2) \mathbb{W} w_3$ . The result of  $(w_1 \mathbb{W} w_2)$  is a language which is weakly parallel composed to a word, i.e. we have to define the weak parallel composition of a language and a word. For the same reason we also need the definition of the weak parallel composition of words (Definition 39) on the level of languages, which is obtained by a new lemma.

**Definition 44. Weakly parallel composition of a languages with a word**

Let  $w \in (X_*)^*$  and  $L \subseteq (X_*)^*$ . The weak parallel composition of a language with a word is given by

$$L \mathbb{W} w = L \mathbb{W} \{w\} \text{ and } w \mathbb{W} L = \{w\} \mathbb{W} L.$$

Lemma 45 transfers the definition of the weak parallel composition of two words (Definition 39) to the level of languages.

**Lemma 45.** Let  $L, L_1$ , and  $L_2$  be languages over  $(X_*)^*$  and  $x, y \in X_*$ . Then it holds

- (i)  $L \mathbb{W} \{\lambda\} = L$ ,
- (ii)  $(\{x\}L_1) \mathbb{W} (\{y\}L_2) = \{x+y\}(L_1 \mathbb{W} L_2) \cup \{x\}(L_1 \mathbb{W} (\{y\}L_2)) \cup \{y\}((\{x\}L_1) \mathbb{W} L_2)$ .

*Proof.*

$$\begin{aligned} \text{(i) } L \mathbb{W} \{\lambda\} &= \bigcup_{w_1 \in L, \lambda \in \{\lambda\}} (w_1 \mathbb{W} \lambda) && \text{[def. 40]} \\ &= \bigcup_{w_1 \in L, \lambda \in \{\lambda\}} \{w_1\} && \text{[def. 39 (i)]} \\ &= L. \end{aligned}$$

$$\begin{aligned} \text{(ii) } (\{x\}L_1) \mathbb{W} (\{y\}L_2) &= \bigcup_{xw_1 \in \{x\}L_1, yw_2 \in \{y\}L_2} (xw_1 \mathbb{W} yw_2) && \text{[def. 40]} \\ &= \bigcup_{w_1 \in L_1, w_2 \in L_2} (xw_1 \mathbb{W} yw_2) \\ &= \bigcup_{w_1 \in L_1, w_2 \in L_2} (\{x+y\} (w_1 \mathbb{W} w_2) \cup \{x\} (w_1 \mathbb{W} (yw_2)) \cup \{y\} ((xw_1) \mathbb{W} w_2)) && \text{[def. 39 (ii)]} \\ &= \bigcup_{w_1 \in L_1, w_2 \in L_2} (\{x+y\} (w_1 \mathbb{W} w_2) \cup \bigcup_{w_1 \in L_1, w_2 \in L_2} \{x\} (w_1 \mathbb{W} (yw_2)) \cup \bigcup_{w_1 \in L_1, w_2 \in L_2} \{y\} ((xw_1) \mathbb{W} w_2)) \end{aligned}$$

$$\begin{aligned}
&= \{x+y\} \bigcup_{w_1 \in L_1, w_2 \in L_2} (w_1 \mathbb{W} w_2) \cup \{x\} \bigcup_{w_1 \in L_1, yw_2 \in \{y\}L_2} (w_1 \mathbb{W} (yw_2)) \cup \{y\} \bigcup_{xw_1 \in \{x\}L_1, w_2 \in L_2} ((xw_1) \mathbb{W} w_2) \\
&= \{x+y\} (L_1 \mathbb{W} L_2) \cup \{x\} (L_1 \mathbb{W} (\{y\}L_2)) \cup \{y\} ((\{x\}L_1) \mathbb{W} L_2). \quad [\text{def. 40}]
\end{aligned}$$

□

Now we have all preconditions to prove Proposition 42.

*Proof.*

### 1. Commutativity

a)  $w_1 \mathbb{W} w_2 = w_2 \mathbb{W} w_1$  is proved by induction over  $|w_1 + w_2|$ .

*Basis:*  $|w_1| + |w_2| = 0$ :

$|w_1| + |w_2| = 0 \Leftrightarrow w_1 = \lambda$  and  $w_2 = \lambda$ .

Hence, we obtain:  $w_1 \mathbb{W} w_2 = \lambda \mathbb{W} \lambda \stackrel{39(i)}{=} \{\lambda\} \stackrel{39(i)}{=} \lambda \mathbb{W} \lambda = w_2 \mathbb{W} w_1$ .

*Hypothesis:*  $w_1 \mathbb{W} w_2 = w_2 \mathbb{W} w_1$  for all  $w_1, w_2$  with  $|w_1| + |w_2| < n, n \in \mathbb{N}$ .

*Step:* Proof for  $|w'_1| + |w'_2| = n, n \geq 1$ .

1.  $w'_1 = \lambda$ :

$\lambda \mathbb{W} w'_2 \stackrel{39(i)}{=} \{w'_2\} \stackrel{39(i)}{=} w'_2 \mathbb{W} \lambda$ .

2.  $w'_2 = \lambda$ :

analogously.

3.  $w'_1 = x_1 w_1, w_2 = x_2 w_2$ :

$$\begin{aligned}
&(x_1 w_1) \mathbb{W} (x_2 w_2) \\
&= \{x_1+x_2\} (w_1 \mathbb{W} w_2) \cup \{x_1\} (w_1 \mathbb{W} (x_2 w_2)) \cup \{x_2\} ((x_1 w_1) \mathbb{W} w_2) \quad [\text{def. 39 (ii)}] \\
&= \{x_1+x_2\} (w_2 \mathbb{W} w_1) \cup \{x_1\} ((x_2 w_2) \mathbb{W} (w_1)) \cup \{x_2\} (w_2 \mathbb{W} (x_1 w_1)) \quad [\text{i.h.}] \\
&= \{x_2+x_1\} (w_2 \mathbb{W} w_1) \cup \{x_1\} ((x_2 w_2) \mathbb{W} (w_1)) \cup \{x_2\} (w_2 \mathbb{W} (x_1 w_1)) \quad [\text{comm. +}] \\
&= ((x_2 w_2) \mathbb{W} (x_1 w_1)). \quad [\text{def. 39 (ii)}]
\end{aligned}$$

b)  $w_1 \$ w_2 = w_2 \$ w_1$  is proved by induction over the structure of  $w_1$ .

*Basis:*  $w_1 = \lambda$ :

$\lambda \$ w_2 \stackrel{39(i)}{=} w_2 \stackrel{39(i)}{=} w_2 \$ \lambda$ .

*Hypothesis:*  $w_1 \$ w_2 = w_2 \$ w_1$  for  $w_1$  arbitrary but fixed and for all  $w_2$ .

*Step:* Proof for  $xw_1$  for all  $x \in X_*$ .

1.  $w_2 = \lambda$ :

$$(xw_1)\$ \lambda \stackrel{39(i)}{=} xw_1 \stackrel{39(i)}{=} \lambda\$ (xw_1).$$

2.  $w_2 = yw'_2$ :

$$\begin{aligned} (xw_1)\$(yw'_2) &= (x+y)(w_1\$w'_2) && [\text{def. 39 (ii)}] \\ &= (x+y)(w'_2\$w_1) && [\text{i.h.}] \\ &= (y+x)(w'_2\$w_1) && [\text{comm. +}] \\ &= (yw'_2)\$(xw_1). && [\text{def. 39 (ii)}] \end{aligned}$$

## 2. Associativity

a)  $(w_1 \mathbb{W} w_2) \mathbb{W} w_3 = w_1 \mathbb{W} (w_2 \mathbb{W} w_3)$  is proved by induction over  $|w_1| + |w_2| + |w_3|$ .

*Basis:*  $|w_1| + |w_2| + |w_3| = 0$  :

$|w_1| + |w_2| + |w_3| = 0 \Leftrightarrow w_1 = \lambda, w_2 = \lambda, w_3 = \lambda$ , and hence

$$(\lambda \mathbb{W} \lambda) \mathbb{W} \lambda \stackrel{39(i)}{=} \{\lambda\} \mathbb{W} \lambda \stackrel{44}{=} \{\lambda\} \mathbb{W} \{\lambda\} \stackrel{44}{=} \lambda \mathbb{W} \{\lambda\} \stackrel{39(i)}{=} \lambda \mathbb{W} (\lambda \mathbb{W} \lambda).$$

*Hypothesis:*  $(w_1 \mathbb{W} w_2) \mathbb{W} w_3 = w_1 \mathbb{W} (w_2 \mathbb{W} w_3)$  for all  $w_1, w_2, w_3$  with  $|w_1| + |w_2| + |w_3| < n, n \in \mathbb{N}$ .

*Step:* Proof for  $|w'_1| + |w'_2| + |w'_3| = n, n \geq 1$ .

1.  $w'_1 = \lambda$ :

$$(\lambda \mathbb{W} w'_2) \mathbb{W} w'_3 \stackrel{39(i)}{=} (w'_2) \mathbb{W} w'_3 \stackrel{39(i)}{=} (w'_2) \mathbb{W} (w'_3 \mathbb{W} \lambda).$$

2.  $w'_2 = \lambda$ :

analogously to 1.

3.  $w'_3 = \lambda$ :

analogously to 1.

4.  $w'_1 = x\bar{w}_1, w'_2 = y\bar{w}_2, w'_3 = z\bar{w}_3$  with  $x, y, z \in X_*$ :

(We use different colours for  $x\bar{w}_1$ ,  $y\bar{w}_2$ , and  $z\bar{w}_3$  in order to reenact how they are de- and recomposed during the proof. Moreover, we label expressions by numbers [1], [2], etc. in order to keep the proof readable. To indicate by which number an expression is represented by, we write [1], [2], etc. next to the respective expression. The use of the label instead of the whole expression is indicated by [1], [2], etc..)

$$\begin{aligned} & ((x\bar{w}_1) \mathbb{W} (y\bar{w}_2)) \mathbb{W} (z\bar{w}_3) \\ &= (\{x+y\}(\bar{w}_1 \mathbb{W} \bar{w}_2) \cup \{x\}(\bar{w}_1 \mathbb{W} (y\bar{w}_2)) \cup \{y\}((x\bar{w}_1) \mathbb{W} \bar{w}_2)) \mathbb{W} (z\bar{w}_3) \quad [\text{def. 39}] \end{aligned}$$

$$\begin{aligned}
&= (\{x+y\}(\bar{w}_1 \mathbb{W} \bar{w}_2) \cup \{x\}(\bar{w}_1 \mathbb{W} (y\bar{w}_2)) \cup \{y\}((x\bar{w}_1) \mathbb{W} \bar{w}_2)) \mathbb{W} \{z\bar{w}_3\} \quad [\text{def. 44}] \\
&= (\{x+y\}(\bar{w}_1 \mathbb{W} \bar{w}_2)) \mathbb{W} \{z\bar{w}_3\} \cup \quad [\text{prop. 43 (3.a)}] \\
&\quad (\{x\}(\bar{w}_1 \mathbb{W} (y\bar{w}_2))) \mathbb{W} \{z\bar{w}_3\} \cup \\
&\quad (\{y\}((x\bar{w}_1) \mathbb{W} \bar{w}_2)) \mathbb{W} \{z\bar{w}_3\} \\
&= (\{x+y\}(\bar{w}_1 \mathbb{W} \bar{w}_2)) \mathbb{W} (\{z\}\{\bar{w}_3\}) \cup \\
&\quad (\{x\}(\bar{w}_1 \mathbb{W} (y\bar{w}_2))) \mathbb{W} (\{z\}\{\bar{w}_3\}) \cup \\
&\quad (\{y\}((x\bar{w}_1) \mathbb{W} \bar{w}_2)) \mathbb{W} (\{z\}\{\bar{w}_3\}) \\
&= \{(x+y)+z\}((\bar{w}_1 \mathbb{W} \bar{w}_2) \mathbb{W} \{\bar{w}_3\})[1] \cup \quad [\text{lem. 45}] \\
&\quad \{x+y\}((\bar{w}_1 \mathbb{W} \bar{w}_2) \mathbb{W} (\{z\}\{\bar{w}_3\})) [2] \cup \\
&\quad \{z\}(\{x+y\}(\bar{w}_1 \mathbb{W} \bar{w}_2)) \mathbb{W} \{\bar{w}_3\} \cup \\
&\quad \{x+z\}((\bar{w}_1 \mathbb{W} (y\bar{w}_2)) \mathbb{W} \{\bar{w}_3\}) [3] \cup \\
&\quad \{x\}((\bar{w}_1 \mathbb{W} (y\bar{w}_2)) \mathbb{W} (\{z\}\{\bar{w}_3\})) [4] \cup \\
&\quad \{z\}(\{x\}(\bar{w}_1 \mathbb{W} (y\bar{w}_2))) \mathbb{W} \{\bar{w}_3\} \cup \\
&\quad \{y+z\}((x\bar{w}_1) \mathbb{W} \bar{w}_2) \mathbb{W} \{\bar{w}_3\} [5] \cup \\
&\quad \{y\}(((x\bar{w}_1) \mathbb{W} \bar{w}_2) \mathbb{W} (\{z\}\{\bar{w}_3\})) [6] \cup \\
&\quad \{z\}(\{y\}((x\bar{w}_1) \mathbb{W} \bar{w}_2)) \mathbb{W} \{\bar{w}_3\} \\
&= [1] \cup [2] \cup [3] \cup [4] \cup [5] \cup [6] \cup \quad [\text{distr. } \circ \text{ over } \cup] \\
&\quad \{z\}(\{x+y\}(\bar{w}_1 \mathbb{W} \bar{w}_2)) \mathbb{W} \{\bar{w}_3\} \cup (\{x\}(\bar{w}_1 \mathbb{W} (y\bar{w}_2))) \mathbb{W} \{\bar{w}_3\} \cup (\{y\}((x\bar{w}_1) \mathbb{W} \bar{w}_2)) \mathbb{W} \{\bar{w}_3\} \\
&= [1] \cup [2] \cup [3] \cup [4] \cup [5] \cup [6] \cup \quad [\text{distr. } \mathbb{W} \text{ over } \cup \text{ (prop. 43 (3))}] \\
&\quad \{z\}(\{x+y\}(\bar{w}_1 \mathbb{W} \bar{w}_2) \cup \{x\}(\bar{w}_1 \mathbb{W} (y\bar{w}_2)) \cup \{y\}((x\bar{w}_1) \mathbb{W} \bar{w}_2)) \mathbb{W} \{\bar{w}_3\} \\
&= [1] \cup [2] \cup [3] \cup [4] \cup [5] \cup [6] \cup \quad [\text{def. 39 (ii)}] \\
&\quad \{z\}(\{(x\bar{w}_1) \mathbb{W} (y\bar{w}_2)\} \mathbb{W} \{\bar{w}_3\}) \\
&= \{x+(y+z)\}(\bar{w}_1 \mathbb{W} (\bar{w}_2 \mathbb{W} \{\bar{w}_3\})) [1] \cup \quad [\text{i.h., ass. +}] \\
&\quad \{x+y\}(\bar{w}_1 \mathbb{W} (\bar{w}_2 \mathbb{W} (\{z\}\{\bar{w}_3\}))) [2] \cup \\
&\quad \{x+z\}(\bar{w}_1 \mathbb{W} ((y\bar{w}_2) \mathbb{W} \{\bar{w}_3\})) [3] \cup \\
&\quad \{x\}(\bar{w}_1 \mathbb{W} ((y\bar{w}_2) \mathbb{W} (\{z\}\{\bar{w}_3\}))) [4] \cup \\
&\quad \{y+z\}((x\bar{w}_1) \mathbb{W} (\bar{w}_2 \mathbb{W} \{\bar{w}_3\})) [5] \cup \\
&\quad \{y\}(((x\bar{w}_1) \mathbb{W} (\bar{w}_2 \mathbb{W} (\{z\}\{\bar{w}_3\})))) [6] \cup \\
&\quad \{z\}((x\bar{w}_1) \mathbb{W} ((y\bar{w}_2) \mathbb{W} \{\bar{w}_3\})) [7] \\
&= [1] \cup [2] \cup [3] \cup \{x\}(\bar{w}_1 \mathbb{W} ((y\bar{w}_2) \mathbb{W} (z\bar{w}_3))) \cup [5] \cup [6] \cup [7] \quad [\text{def. 44}] \\
&= [1] \cup [2] \cup [3] \cup \quad [\text{def. 39 (ii)}] \\
&\quad \{x\}(\bar{w}_1 \mathbb{W} (\{y+z\}(\bar{w}_2 \mathbb{W} \bar{w}_3) \cup \{y\}(\bar{w}_2 \mathbb{W} (z\bar{w}_3)) \cup \{z\}((y\bar{w}_2) \mathbb{W} \bar{w}_3))) \cup \\
&\quad [5] \cup [6] \cup [7] \\
&= [1] \cup [2] \cup [3] \cup \quad [\text{distr. } \mathbb{W} \text{ over } \cup] \\
&\quad \{x\}(\bar{w}_1 \mathbb{W} (\{y+z\}(\bar{w}_2 \mathbb{W} \bar{w}_3)) \cup \bar{w}_1 \mathbb{W} (\{y\}(\bar{w}_2 \mathbb{W} (z\bar{w}_3))) \cup \bar{w}_1 \mathbb{W} (\{z\}((y\bar{w}_2) \mathbb{W} \bar{w}_3))) \cup \\
&\quad [5] \cup [6] \cup [7] \\
&= [1] \cup [2] \cup [3] \cup \quad [\text{distr. } \circ \text{ over } \cup] \\
&\quad \{x\}(\bar{w}_1 \mathbb{W} (\{y+z\}(\bar{w}_2 \mathbb{W} \bar{w}_3))) [4a] \cup \quad \{x\}(\bar{w}_1 \mathbb{W} (\{y\}(\bar{w}_2 \mathbb{W} (z\bar{w}_3)))) [4b] \cup \\
&\quad \{x\}(\bar{w}_1 \mathbb{W} (\{z\}((y\bar{w}_2) \mathbb{W} \bar{w}_3))) [4c] \cup \\
&\quad [5] \cup [6] \cup [7]
\end{aligned}$$

$$\begin{aligned}
&= \{x+(y+z)\}(\bar{w}_1 \mathbb{W}(\bar{w}_2 \mathbb{W}\{\bar{w}_3\})) [1] \cup && [\text{ass. } \cup] \\
&\{x\}(\bar{w}_1 \mathbb{W}(\{y+z\}(\bar{w}_2 \mathbb{W}\bar{w}_3))) [4a] \cup \\
&\{y+z\}((x\bar{w}_1) \mathbb{W}(\bar{w}_2 \mathbb{W}\{\bar{w}_3\})) [5] \cup \\
&\{x+y\}(\bar{w}_1 \mathbb{W}(\bar{w}_2 \mathbb{W}(\{z\}\{\bar{w}_3\}))) [2] \cup \\
&\{x\}(\bar{w}_1 \mathbb{W}(\{y\}(\bar{w}_2 \mathbb{W}(z\bar{w}_3)))) [4b] \cup \\
&\{y\}(((xw_1) \mathbb{W}(\bar{w}_2 \mathbb{W}(\{z\}\{\bar{w}_3\})))) [6] \cup \\
&\{x+z\}(\bar{w}_1 \mathbb{W}((y\bar{w}_2) \mathbb{W}\{\bar{w}_3\})) [3] \cup \\
&\{x\}(\bar{w}_1 \mathbb{W}(\{z\}((y\bar{w}_2) \mathbb{W}\bar{w}_3))) [4c] \cup \\
&\{z\}((x\bar{w}_1) \mathbb{W}((y\bar{w}_2) \mathbb{W}\{\bar{w}_3\})) [7] \\
&= (\{x\}\{\bar{w}_1\}) \mathbb{W}(\{y+z\}(\bar{w}_2 \mathbb{W}\bar{w}_3)) \cup && [\text{lem. 45}] \\
&(\{x\}\{\bar{w}_1\}) \mathbb{W}(\{y\}(\bar{w}_2 \mathbb{W}(z\bar{w}_3))) \cup \\
&(\{x\}\{\bar{w}_1\}) \mathbb{W}(\{z\}((y\bar{w}_2) \mathbb{W}\bar{w}_3)) \\
&= (\{x\}\{\bar{w}_1\}) \mathbb{W}(\{y+z\}(\bar{w}_2 \mathbb{W}\bar{w}_3) \cup \{y\}(\bar{w}_2 \mathbb{W}(z\bar{w}_3)) \cup \{z\}((y\bar{w}_2) \mathbb{W}\bar{w}_3)) \\
&= \{x\bar{w}_1\} \mathbb{W}((y\bar{w}_2) \mathbb{W}(z\bar{w}_3)) && [\text{def. 39 (ii)}] \\
&= (x\bar{w}_1) \mathbb{W}((y\bar{w}_2) \mathbb{W}(z\bar{w}_3)). && [\text{def. 44}]
\end{aligned}$$

b)  $(w_1\$w_2)\$w_3 = w_1\$(w_2\$w_3)$  is proved by induction over  $|w_1| + |w_2| + |w_3|$ .

$$\begin{aligned}
&\text{Basis: } |w_1| + |w_2| + |w_3| = 0 : \\
&|w_1| + |w_2| + |w_3| = 0 \Leftrightarrow w_1 = w_2 = w_3 = \lambda. \\
&(\lambda\$ \lambda)\$ \lambda \stackrel{39(i)}{=} \lambda\$ \lambda \stackrel{39(i)}{=} \lambda\$(\lambda\$ \lambda).
\end{aligned}$$

*Hypothesis:*  $(w_1\$w_2)\$w_3 = w_1\$(w_2\$w_3)$  for all  $w_1, w_2, w_3$  with  $|w_1| + |w_2| + |w_3| < n, n \in \mathbb{N}$ .

*Step:* Proof for  $|w'_1| + |w'_2| + |w'_3| = n$ .

$$\begin{aligned}
&1. w'_1 = \lambda: \\
&(\lambda\$w_2)\$w_3 \stackrel{39(i)}{=} w_2\$w_3 \stackrel{39(i)}{=} \lambda\$(w_2\$w_3).
\end{aligned}$$

2.  $w'_2 = \lambda$ :  
analogously to 1.

3.  $w'_3 = \lambda$ :  
analogously to 1.

$$\begin{aligned}
&4. w'_1 = x\bar{w}_1, w'_2 = y\bar{w}_2, w'_3 = z\bar{w}_3 \text{ with } x, y, z \in X_*: \\
&((x\bar{w}_1)\$(y\bar{w}_2))\$(z\bar{w}_3) = ((x+y)(\bar{w}_1\$\bar{w}_2))\$(z\bar{w}_3) && [\text{def. 39(i)}] \\
&= ((x+y)+z)((\bar{w}_1\$\bar{w}_2)\$\bar{w}_3) && | \text{def. 39(i)} \\
&= (x+(y+z))(\bar{w}_1\$(\bar{w}_2\$\bar{w}_3)) && [\text{i.h., ass. +}]
\end{aligned}$$

$$\begin{aligned}
&= (x\bar{w}_1)\$(y+z)(\bar{w}_2\$\bar{w}_3) && \text{[def. 39(i)]} \\
&= (x\bar{w}_1)\$(y\bar{w}_2)\$(z\bar{w}_3). && \text{[def. 39(i)]}
\end{aligned}$$

□

The proof for Proposition 43 (properties of the parallel operators for languages) is based on the respective properties of the operators for words.

*Proof.* Properties of  $\mathbb{W}$  and  $\$$  for languages (Proposition 43)

### 1. Commutativity

$$\begin{aligned}
\text{a) } L_1 \mathbb{W} L_2 &= \bigcup_{w_1 \in L_1, w_2 \in L_2} w_1 \mathbb{W} w_2 && \text{[def. 40]} \\
&= \bigcup_{w_1 \in L_1, w_2 \in L_2} w_2 \mathbb{W} w_1 && \text{[prop. 42 (1)]} \\
&= L_2 \mathbb{W} L_1. && \text{[def. 40]}
\end{aligned}$$

$$\begin{aligned}
\text{b) } L_1 \$ L_2 &= \{w_1 \$ w_2 \mid w_1 \in L_1, w_2 \in L_2\} && \text{[def. 40]} \\
&= \{w_2 \$ w_1 \mid w_1 \in L_1, w_2 \in L_2\} && \text{[prop. 42 (1)]} \\
&= L_2 \$ L_1. && \text{[def. 40]}
\end{aligned}$$

### 2. Associativity

$$\begin{aligned}
\text{a) } (L_1 \mathbb{W} L_2) \mathbb{W} L_3 &= \left( \bigcup_{w_1 \in L_1, w_2 \in L_2} w_1 \mathbb{W} w_2 \right) \mathbb{W} L_3 && \text{[def. 40]} \\
&= \bigcup_{w_1 \in L_1, w_2 \in L_2, w_3 \in L_3} (w_1 \mathbb{W} w_2) \mathbb{W} w_3 && \text{[def. 40]} \\
&= \bigcup_{w_1 \in L_1, w_2 \in L_2, w_3 \in L_3} (w_1 \mathbb{W} (w_2 \mathbb{W} w_3)) && \text{[prop. 42 (2)]} \\
&= L_1 \mathbb{W} \left( \bigcup_{w_2 \in L_2, w_3 \in L_3} w_2 \mathbb{W} w_3 \right) && \text{[def. 40]} \\
&= L_1 \mathbb{W} (L_2 \mathbb{W} L_3). && \text{[def. 40]}
\end{aligned}$$

$$\begin{aligned}
\text{b) } (L_1 \$ L_2) \$ L_3 &= \{w_1 \$ w_2 \mid w_1 \in L_1, w_2 \in L_2\} \$ L_3 && \text{[def. 40]} \\
&= \{(w_1 \$ w_2) \$ w_3 \mid w_1 \in L_1, w_2 \in L_2, w_3 \in L_3\} && \text{[def. 40]} \\
&= \{w_1 \$ (w_2 \$ w_3) \mid w_1 \in L_1, w_2 \in L_2, w_3 \in L_3\} && \text{[prop. 42 (2)]} \\
&= L_1 \$ \{w_2 \$ w_3 \mid w_2 \in L_2, w_3 \in L_3\} && \text{[def. 40]} \\
&= L_1 \$ (L_2 \$ L_3). && \text{[def. 40]}
\end{aligned}$$

### 3. Right distributivity over $\cup$

$$\begin{aligned}
\text{a) } (L_1 \cup L_2) \mathbb{W} L_3 &= \bigcup_{w'_1 \in L_1 \cup L_2, w_3 \in L_3} w'_1 \mathbb{W} w_3 && \text{[def. 40]} \\
&= \bigcup_{w_1 \in L_1, w_3 \in L_3} w_1 \mathbb{W} w_3 \cup \bigcup_{w_2 \in L_2, w_3 \in L_3} w_2 \mathbb{W} w_3 \\
&= (L_1 \mathbb{W} L_3) \cup (L_2 \mathbb{W} L_3). && \text{[def. 40]}
\end{aligned}$$

$$\text{b) } (L_1 \cup L_2) \$ L_3 = \{w \$ w_3 \mid w \in L_1 \cup L_2, w_3 \in L_3\} \quad \text{[def. 40]}$$

$$\begin{aligned}
&= \{w_1\$w_3 \mid w_1 \in L_1, w_3 \in L_3\} \cup \{w_2\$w_3 \mid w_2 \in L_2, w_3 \in L_3\} \\
&= L_2\$L_3 \cup L_2\$L_3. \qquad \qquad \qquad \text{[def. 40]}
\end{aligned}$$

#### 4. Left distributivity over $\cup$

The proof is analogous to 3.

#### 5. $\emptyset$ is annihilator

$$\text{a) } L\mathbb{W}\emptyset \stackrel{40}{=} \bigcup_{w_1 \in L, w_2 \in \emptyset} w_1\mathbb{W}w_2 = \emptyset.$$

$$\text{b) } L\$ \emptyset \stackrel{40}{=} \{w_1\$w_2 \mid w_1 \in L, w_2 \in \emptyset\} = \emptyset.$$

#### 6. $\{\lambda\}$ is identity

$$\text{a) } L\mathbb{W}\{\lambda\} \stackrel{40}{=} \bigcup_{w_1 \in L, w_2 \in \{\lambda\}} w_1\mathbb{W}w_2 = \bigcup_{w_1 \in L} w_1\mathbb{W}\lambda \stackrel{39(i)}{=} \bigcup_{w_1 \in L} w_1 = L.$$

$$\begin{aligned}
\text{b) } L\$ \lambda &= \{w_1\$w_2 \mid w_1 \in L, w_2 \in \{\lambda\}\} && \text{[def. 40]} \\
&= \{w_1\$ \lambda \mid w_1 \in L\} \\
&= \{w_1 \mid w_1 \in L\} && \text{[def. 39(i)]} \\
&= L. && \square
\end{aligned}$$

### 3.2.3 Algebraic Laws for Weak and Synchronous Expressions

The last section has introduced weak parallel and synchronous composition of words and languages and their properties. This justifies to formulate the properties as algebraic laws on the syntactic level of parallel expressions.

#### Definition 46. Algebraic laws

Let  $C, C_1, C_2$  and  $C_3$  be weak resp. synchronous expressions and  $r_1, r_2 \in X_*$ . For the composition operators  $\mathbb{W}$  and  $\$$ , denoted summarising by  $\parallel$ , the following algebraic laws hold:

1.  $\emptyset$  is the annihilator for parallel composition  
 $\emptyset \parallel C = \emptyset.$
2.  $\lambda$  is the identity for parallel composition  
 $\lambda \parallel C = C.$
3. Commutative law for parallel composition  
 $C_1 \parallel C_2 = C_2 \parallel C_1.$
4. Associative law for parallel composition  
 $(C_1 \parallel C_2) \parallel C_3 = C_1 \parallel (C_2 \parallel C_3).$

5. Right distributive law of parallel composition over choice  
 $(C_1 | C_2) \parallel C_3 = (C_1 \parallel C_3) | (C_2 \parallel C_3)$ .
6. Left distributive law of parallel composition over choice  
 $C_1 \parallel (C_2 | C_3) = (C_1 \parallel C_2) | (C_1 \parallel C_3)$ .
7. Dissolving  $\mathbb{W}$  to  $+$ 
  - (i)  $(r_1; C_1) \mathbb{W} (r_2; C_2) = r_1+r_2; (C_1 \mathbb{W} C_2) | r_1; (C_1 \mathbb{W} (r_2; C_2)) | r_2; ((r_1; C_1) \mathbb{W} C_2)$ ,
  - (ii)  $r_1 \mathbb{W} (r_2; C_2) = r_1+r_2; C_2 | r_1; r_2; C_2 | r_2; (r_1 \mathbb{W} C_2)$ ,
  - (iii)  $r_1 \mathbb{W} r_2 = r_1+r_2 | r_1; r_2 | r_2; r_1$ .
8. Dissolving  $\$$  to  $+$ 
  - (i)  $(r_1; C_1) \$ (r_2; C_2) = (r_1 + r_2); (C_1 \$ C_2)$ ,
  - (ii)  $r_1 \$ (r_2; C_2) = (r_1 + r_2); C_2$ ,
  - (iii)  $r_1 \$ r_2 = (r_1 + r_2)$ .

The algebraic laws for regular expressions still apply for parallel expressions. In order to reference the respective laws when transforming parallel expressions we list the employed laws in the following:

9.  $\emptyset; C = \emptyset = C; \emptyset$ ,
10.  $\emptyset | C = C$ ,
11.  $\emptyset^* = \lambda$ ,
12.  $\lambda; C = C = C; \lambda$ ,
13.  $\lambda^* = \lambda$ ,
14.  $(C_1 | C_2); C_3 = C_1; C_3 | C_2; C_3$ ,
15.  $C_1; (C_2 | C_3) = C_1; C_2 | C_1; C_3$ ,
16.  $C^* = (\lambda | C; C^*)$  if  $C$  is not of the form  $C'^*$ ,  $(\lambda | C')$ , or  $\lambda$ . Otherwise one of the following laws has to be applied first.
  - a)  $(C^*)^* = C^*$ ,
  - b)  $\lambda^* = \lambda$ ,
  - c)  $(\lambda | C)^* = C^*$ .

In 16. we restrict the application of the law  $C^* = (\lambda | C; C^*)$  in order to avoid repetitions of the original expressions when resolving it. Consider, e.g., the expression  $(\lambda | r)^* \parallel r'$ . By applying the algebraic laws (without restrictions) we obtain



$$\begin{aligned}
& (\lambda \mid r)^* \parallel r' = (\lambda \mid (\lambda \mid r); (\lambda \mid r)^*) \parallel r' \\
& = \lambda \parallel r' \mid (\lambda \mid r); (\lambda \mid r)^* \parallel r' \\
& = \lambda \parallel r' \mid (\lambda; (\lambda \mid r)^* \mid r; (\lambda \mid r)^*) \parallel r' \\
& = \lambda \parallel r' \mid \lambda; (\lambda \mid r)^* \parallel r' \mid r; (\lambda \mid r)^* \parallel r' \\
& = r' \mid \underline{(\lambda \mid r)^* \parallel r' \mid r; (\lambda \mid r)^* \parallel r'}.
\end{aligned}$$

The second of the resulting alternative expressions is the same as the original one.

With the restrictions we have made we obtain the equations

$$\begin{aligned}
& (\lambda \mid r)^* \parallel r' = r^* \parallel r' \\
& = (\lambda \mid r; r^*) \parallel r' \\
& = \lambda \parallel r' \mid (r; r^*) \parallel r' \\
& = r' \mid (r; r^*) \parallel r' \\
& = r' \mid r + r'; r^*,
\end{aligned}$$

which contain no repetition of the original expression.

In the following we state and prove that the algebraic laws introduced in Definition 46 preserve the language of parallel expressions, i.e. if a parallel expression  $C$  is transformed to  $C'$  by application of some algebraic laws the language of both expressions is still the same.

**Proposition 47.** Let  $C = C'$  be an algebraic law. Then  $L(C) = L(C')$ .

*Proof.*

Let  $C, C_1, C_2, C_3$  be parallel expressions.

1.  $\emptyset \parallel C = \emptyset$ :
  - a)  $L(\emptyset \parallel C) \stackrel{41}{=} L(\emptyset) \parallel L(C) \stackrel{41}{=} \emptyset \parallel L(C) \stackrel{43-5}{=} \emptyset \stackrel{41}{=} L(\emptyset)$ .
  - b)  $L(\emptyset \$ C) = L(\emptyset) \$ L(C) = \emptyset \$ L(C) = \emptyset = L(\emptyset)$ .
2.  $\lambda \parallel C = C$ :
  - a)  $L(\lambda \parallel C) = L(\lambda) \parallel L(C)$  [def. 41]
    - $= \{\lambda\} \parallel L(C)$  [def. 41]
    - $= \bigcup_{w_1 \in \{\lambda\}, w_2 \in L(C)} w_1 \parallel w_2$  [def. 39]
    - $= \bigcup_{w_2 \in L(C)} \lambda \parallel w_2$
    - $= \bigcup_{w_2 \in L(C)} w_2$  [def. 39]
    - $= L(C)$ .

$$\begin{aligned}
\text{b) } L(\lambda\$C) &= L(\lambda)\$L(C) \\
&= \{\lambda\}\$L(C) \\
&= \{w_1\$w_2 \mid w_1 \in \{\lambda\}, w_2 \in L(C)\} \\
&= \{\lambda\$w_2 \mid w_2 \in L(C)\} \\
&= \{w_2 \mid w_2 \in L(C)\} \\
&= L(C).
\end{aligned}$$

$$\begin{aligned}
3. \ C_1 \parallel C_2 &= C_2 \parallel C_1: \\
\text{a) } L(C_1 \mathbb{W} C_2) &\stackrel{41-7a}{=} L(C_1) \mathbb{W} L(C_2) \stackrel{43-1}{=} L(C_2) \mathbb{W} L(C_1) \stackrel{41-7a}{=} L(C_2 \mathbb{W} C_1).
\end{aligned}$$

$$\text{b) } L(C_1\$C_2) = L(C_1)\$L(C_2) = L(C_2)\$L(C_1) = L(C_2\$C_1).$$

$$\begin{aligned}
4. \ (C_1 \parallel C_2) \parallel C_3 &= C_1 \parallel (C_2 \parallel C_3): \\
\text{a) } L((C_1 \mathbb{W} C_2) \mathbb{W} C_3) &= L(C_1 \mathbb{W} C_2) \mathbb{W} L(C_3) && [\text{def. 41 (7a)}] \\
&= (L(C_1) \mathbb{W} L(C_2)) \mathbb{W} L(C_3) && [\text{def. 41 (7a)}] \\
&= L(C_1) \mathbb{W} (L(C_2) \mathbb{W} L(C_3)) && [\text{prop. 43 - 2}] \\
&= L(C_1) \mathbb{W} (L(C_2 \mathbb{W} C_3)) && [\text{def. 41 (7a)}] \\
&= L(C_1 \mathbb{W} (C_2 \mathbb{W} C_3)). && [\text{def. 41 (7a)}]
\end{aligned}$$

$$\begin{aligned}
\text{b) } L((C_1\$C_2)\$C_3) &= L(C_1\$C_2)\$L(C_3) \\
&= (L(C_1)\$L(C_2))\$L(C_3) \\
&= L(C_1)\$(L(C_2)\$L(C_3)) \\
&= L(C_1)\$(L(C_2\$C_3)) \\
&= L(C_1\$C_2)\$(C_3).
\end{aligned}$$

$$\begin{aligned}
5. \ (C_1 \mid C_2) \parallel C_3 &= (C_1 \parallel C_3) \mid (C_2 \parallel C_3): \\
\text{a) } L(C_1 \mid C_2) \mathbb{W} C_3 &= L(C_1 \mid C_2) \mathbb{W} L(C_3) && [\text{def. 41 (7a)}] \\
&= (L(C_1) \cup L(C_2)) \mathbb{W} L(C_3) && [\text{def. 41 (5)}] \\
&= (L(C_1) \mathbb{W} L(C_3)) \cup (L(C_2) \mathbb{W} L(C_3)) && [\text{def. 41}] \\
&= (L(C_1 \mathbb{W} C_3)) \cup (L(C_2 \mathbb{W} C_3)) && [\text{def. 41}] \\
&= L((C_1 \mathbb{W} C_3) \mid (C_2 \mathbb{W} C_3)).
\end{aligned}$$

$$\begin{aligned}
\text{b) } L(C_1 \mid C_2)\$C_3 &= L(C_1 \mid C_2)\$L(C_3) \\
&= (L(C_1) \cup L(C_2))\$L(C_3) \\
&= (L(C_1)\$L(C_3)) \cup (L(C_2)\$L(C_3)) \\
&= (L(C_1\$C_3)) \cup (L(C_2\$C_3)) \\
&= L((C_1\$C_3) \mid (C_2\$C_3)).
\end{aligned}$$

$$6. \ C_1 \parallel (C_2 \mid C_3) = (C_1 \parallel C_2) \mid (C_1 \parallel C_3): \text{ analogous to 5.}$$

7. Dissolving  $\mathbb{W}$  to + :

$$\begin{aligned}
(i) \quad L((r_1; C_1) \mathbb{W}(r_2; C_2)) &= L(r_1; C_1) \mathbb{W} L(r_2; C_2) && [\text{def. 41 (7a)}] \\
&= (L(r_1)L(C_1)) \mathbb{W} (L(r_2)L(C_2)) && [\text{def. 41 (4)}] \\
&= \bigcup_{w_1 \in L(r_1)L(C_1), w_2 \in L(r_2)L(C_2)} w_1 \mathbb{W} w_2 && [\text{def. 40}] \\
&= \bigcup_{w'_1 \in L(C_1), w'_2 \in L(C_2)} r_1 w'_1 \mathbb{W} r_2 w'_2 \\
&= \bigcup_{w'_1 \in L(C_1), w'_2 \in L(C_2)} (\{r_1+r_2\}(w'_1 \mathbb{W} w'_2) \cup && [\text{def. 39 (ii)}] \\
&\quad \{r_1\}(w'_1 \mathbb{W} (r_2 w'_2)) \cup \\
&\quad \{r_2\}((r_1 w'_1) \mathbb{W} w'_2)) \\
&= \bigcup_{w'_1 \in L(C_1), w'_2 \in L(C_2)} \{r_1+r_2\}(w'_1 \mathbb{W} w'_2) \cup \\
&\quad \bigcup_{w'_1 \in L(C_1), w'_2 \in L(C_2)} \{r_1\}(w'_1 \mathbb{W} (r_2 w'_2)) \cup \\
&\quad \bigcup_{w'_1 \in L(C_1), w'_2 \in L(C_2)} \{r_2\}((r_1 w'_1) \mathbb{W} w'_2) \\
&= \bigcup_{w'_1 \in L(C_1), w'_2 \in L(C_2)} \{r_1+r_2\}(w'_1 \mathbb{W} w'_2) \cup \\
&\quad \bigcup_{w'_1 \in L(C_1), w'_2 \in L(r_2)L(C_2)} \{r_1\}(w'_1 \mathbb{W} w'_2) \cup \\
&\quad \bigcup_{w'_1 \in L(r_1)L(C_1), w'_2 \in L(C_2)} \{r_2\}(w'_1 \mathbb{W} w'_2) \\
&= \{r_1+r_2\} \bigcup_{w'_1 \in L(C_1), w'_2 \in L(C_2)} (w'_1 \mathbb{W} w'_2) \cup \\
&\quad \{r_1\} \bigcup_{w'_1 \in L(C_1), w'_2 \in L(r_2)L(C_2)} (w'_1 \mathbb{W} w'_2) \cup \\
&\quad \{r_2\} \bigcup_{w'_1 \in L(r_1)L(C_1), w'_2 \in L(C_2)} (w'_1 \mathbb{W} w'_2) \\
&= \{r_1+r_2\} L(C_1) \mathbb{W} L(C_2) \cup && [\text{def.40}] \\
&\quad \{r_1\} L(C_1) \mathbb{W} (L(r_2)L(C_2)) \cup \\
&\quad \{r_2\} (L(r_1)L(C_1)) \mathbb{W} L(C_2) \\
&= L(r_1+r_2)L(C_1) \mathbb{W} L(C_2) \cup && [\text{def. 41 (3)}] \\
&\quad L(r_1)L(C_1) \mathbb{W} (L(r_2)L(C_2)) \cup \\
&\quad L(r_2)(L(r_1)L(C_1)) \mathbb{W} L(C_2) \\
&= L(r_1+r_2)L(C_1 \mathbb{W} C_2) \cup && [\text{def. 41 (7a,4)}] \\
&\quad L(r_1)(L(C_1) \mathbb{W} L(r_2; C_2)) \cup \\
&\quad L(r_2)(L(r_1; C_1)) \mathbb{W} L(C_2) \\
&= L((r_1+r_2); (C_1 \mathbb{W} C_2)) \cup && [\text{def. 41 (4,7a)}] \\
&\quad L(r_1)L(C_1 \mathbb{W} (r_2; C_2)) \cup \\
&\quad L(r_2)L(r_1; C_1) \mathbb{W} C_2 \\
&= L((r_1+r_2); (C_1 \mathbb{W} C_2)) \cup && [\text{def.41 (4)}]
\end{aligned}$$

$$\begin{aligned}
& L(r_1; (C_1 \mathbb{W}(r_2; C_2)) \cup \\
& \quad L(r_2; (r_1; C_1) \mathbb{W} C_2) \\
= & L((r_1+r_2); (C_1 \mathbb{W} C_2) | r_1; (C_1 \mathbb{W}(r_2; C_2)) | r_2; (r_1; C_1) \mathbb{W} C_2). [\text{def. 41 (5)}]
\end{aligned}$$

(ii) follows from 7 (i).

(iii) follows from 7 (i).

8. Dissolving \$ to + :

$$\begin{aligned}
L((r_1; C_1) \$ (r_2; C_2)) &= L(r_1; C_1) \$ L(r_2; C_2) && [\text{def. 41}] \\
&= (L(r_1) L(C_1)) \$ (L(r_2) L(C_2)) \\
&= \{w_1 \$ w_2 \mid w_1 \in L(r_1) L(C_1), w_2 \in L(r_2) L(C_2)\} \\
&= \{r_1 w'_1 \$ r_2 w'_2 \mid w'_1 \in L(C_1), w'_2 \in L(C_2)\} \\
&= \{(r_1+r_2)(w'_1 \$ w'_2) \mid w'_1 \in L(C_1), w'_2 \in L(C_2)\} \\
&= \{r_1+r_2\} L(C_1) \$ L(C_2) \\
&= L(r_1+r_2) L(C_1) \$ L(C_2) \\
&= L(r_1+r_2) L(C_1 \$ C_2) \\
&= L((r_1+r_2); (C_1 \$ C_2)).
\end{aligned}$$

(ii) follows from 8. (i).

(iii) follows from 8. (i).

9.-16. Since items 9-16 are laws for regular expressions the respective proofs already exist (see e.g. [HMU06]).

□

### 3.2.4 Weak and Synchronous Expressions as Control Conditions

We now introduce two approaches how to use parallel expressions as control condition describing permitted derivations. The first approach, called language control, is based on the language of parallel expressions and uses the members of the language as application sequences for the desired derivations. The second approach, called normal form control, constructs the derivations in a stepwise manner employing a canonical form of parallel expressions called *normal form*.

#### Approach 1: Language Control

A parallel expression as control condition is satisfied if the application sequence of a derivation is a member of its corresponding language. i.e. the

semantics of a parallel expression  $C$  used as control condition is given by  $SEM_1(C) = \{G \xRightarrow{*} G' \mid \text{the application sequence } w \text{ of } G \xRightarrow{*} G' \text{ is in } L(C)\}$ .

## Approach 2: Normal Form Control

Another possibility to obtain permitted derivations from a parallel expression is to build them step by step, i.e. for every next upcoming derivation step it has to be identified which rules to apply (in parallel). Recursive constructions as used e.g. in Chapter 5 do not apply here since in general we are not able to compose derivations in parallel without changing the applicability of the involved rules. In order to determine which rules to apply next, we employ a canonical form for parallel expressions, called *normal form*. The normal form of a parallel expression provides all rules that have to be applied in parallel in the next derivation step sequentially composed to the remaining expression. Having applied this rules resulting in one derivation step the remaining expression again can be transformed to normal form and so on. The following section introduces the normal form and provides a procedure how to construct the normal form of a parallel expression. After that the use of a parallel expression as control condition employing the normal form is formally introduced.

### Normal form of parallel expressions

Every parallel expression (except for the special cases  $\emptyset$  and  $\lambda$ ) can be transformed to a choice of expressions given by a parallel rule sequentially composed to some remaining parallel expression, i.e. the rules of the original expression that have to be applied in parallel in the first derivation step are collected to a parallel rule and the remaining expression contains the rest of the original expression.

#### Proposition 48. Normal form

Let  $R$  be a set of rules. Every parallel expression  $C$  over  $R_*$  can be transformed to  $X_1 \mid \dots \mid X_n$  for some  $n \in \mathbb{N}$  with  $X_i \in \{\lambda, r_{i_1} + \dots + r_{i_m}, r_{i_1} + \dots + r_{i_m}; C_i\}$ ,  $i \in [n], m \in \mathbb{N} \setminus \{0\}$ ,  $r_{i_j} \in R_*, j \in [m]$ , and  $C_i$  is a parallel expression.  $X_1 \mid \dots \mid X_n$  is called *normal form* of  $C$ .

As special cases of the normal form we obtain  $\emptyset$  (for  $n = 0$ ),  $\lambda$  (for  $n = 1$  and  $X_1 = \lambda$ ), and  $r$  (for  $n = 1$  and  $X_1 = r$ ). In order to prove Proposition 48 we use the algebraic laws for parallel expressions and regular expressions given in Definition 46.

*Proof.* We prove Proposition 48 by induction over the structure of  $C$ .

Preliminary remark: Since  $(C_1 \parallel C_2) \parallel C_3 = C_1 \parallel (C_2 \parallel C_3)$ , we show in the re-

spective proof steps the proposition only for the structure  $(C_1 \parallel C_2)$ . If e.g.  $C_2$  again is of the form  $C_2' \parallel C_2''$  the proof step has to be repeated and so on. Moreover, since  $C_1 \parallel C_2 = C_2 \parallel C_1$  we only show the proposition for one order. When it comes to the structure of an  $X_i$  we have summarised the cases  $X_i = r_{i_1} + \dots + r_{i_m}$  and  $X_i = r_{i_1} + \dots + r_{i_m}; C_i$  where it makes no difference in the argumentation which case is chosen. In such a summary the  $C_i$  is coloured green and embraced by  $[\ ]$ ,  $r_{i_1} + \dots + r_{i_m} [\ ] ; C_i$ , in order to express that both cases are meant.

*Basis:*

$C = \emptyset$  ✓ for  $n = 0$ ,

$C = \lambda$  ✓ for  $n = 1$  and  $X_1 = \lambda$ ,

$C = r$  ✓ for  $n = 1, m = 1$  and  $X_1 = r_1 = r$ .

*Hypothesis:* Proposition holds for parallel expressions  $C_1, C_2$ , and  $\bar{C}$ .

*Induction step:* Proof for  $C_1; C_2, C_1 \mid C_2, \bar{C}^*, C_1 \parallel C_2$ , and  $C_1 \$ C_2$ .

Let  $o, p, q, l \in \mathbb{N} \setminus \{0\}$

1.  $C = C_1; C_2$

By induction hypothesis we obtain the following cases for the structure of  $C_1$  and  $C_2$

1.1.  $C_1 = \emptyset; C_2 = \emptyset$ . ✓ [def. 46 (9)]

1.2.  $C_2 = \emptyset$ : analogous.

1.3.  $C_1 = \lambda; C_2 = C_2$ . ✓ [def. 46 (12)]

1.4.  $C_2 = \lambda$ : analogous.

1.5.  $C_1 = (Y_1 \mid \dots \mid Y_o), C_2 = (Z_1 \mid \dots \mid Z_p)$ :  
 $(Y_1 \mid \dots \mid Y_o); (Z_1 \mid \dots \mid Z_p) = \mid_{j \in [o], k \in [p]} (Y_j; Z_k)$  [def. 46 (14,15)]

The expression  $\mid_{j \in [o], k \in [p]} (Y_j; Z_k)$  is in normal form if each subexpression  $Y_j; Z_k$

is in normal form.

The following cases for  $Y_j; Z_k$  have to be considered:

1.5.1.  $Y_j = \lambda, Z_k \in \{\lambda, r'_1 + \dots + r'_l [\bar{C}']\}$ :

$\lambda; Z_k = Z_k$  ✓

1.5.2.  $Y_j = (r_1 + \dots + r_q) [\bar{C}], Z_k \in \{\lambda, r'_1 + \dots + r'_l [\bar{C}']\}$ :

$(r_1 + \dots + r_q) [\bar{C}]; Z_k$  ✓

2.  $C = C_1 \mid C_2$

2.1.  $C_1 = \emptyset; C_2 = C_2$ . ✓ due to i.h. [def. 46 (10)]

2.2.  $C_2 = \emptyset$ : analogous.

2.3. For all other forms of  $C_1$  and  $C_2$ :

$C_1 \mid C_2$  is already in the required form due to the induction hypothesis.

3.  $C = \bar{C}^*$

By induction hypothesis we obtain the following cases for the structure of  $\bar{C}$

3.1.  $\bar{C} = \emptyset: \emptyset^* = \lambda. \checkmark$  [def. 46 (11)]

3.2.  $\bar{C} = \lambda: \lambda^* = \lambda. \checkmark$  [def. 46 (13)]

3.3.  $\bar{C} = (Y_1 | \dots | Y_o):$   
 $(Y_1 | \dots | Y_o)^* = |_{i \in o, Y_i \neq \lambda} (Y_i)^*$  [def. 46 (16 c)]

$= (Y_{1'} | \dots | Y_{o'})^*$   
 $= \lambda | (Y_{1'} | \dots | Y_{o'}); (Y_{1'} | \dots | Y_{o'})^*$  [def. 46 (16)]

$= \lambda | |_{i \in o'} (Y_i; (Y_{1'} | \dots | Y_{o'})^*). \checkmark$  [def. 46 (14)]

4.  $C = C_1 \$ C_2$

By induction hypothesis we obtain the following cases for the structure of  $C_1$  and  $C_2$ :

4.1.  $C_1 = \emptyset: \emptyset \$ C_2 = \emptyset. \checkmark$  [def. 46 (1)]

4.2.  $C_1 = \lambda: \lambda \$ C_2 = C_2. \checkmark$  [def. 46 (2)]

4.3.  $C_1 = (Y_1 | \dots | Y_o), C_2 = (Z_1 | \dots | Z_p):$   
 $(Y_1 | \dots | Y_o) \$ (Z_1 | \dots | Z_p) = |_{j \in [o], k \in [p]} (Y_j \$ Z_k)$  [def. 46 (5,6)]

The expression  $|_{j \in [o], k \in [p]} (Y_j \$ Z_k)$  is in normal form if each subexpression

$(Y_j \$ Z_k)$  is in normal form.

4.3.1.  $Y_j = \lambda, Z_k \in \{\lambda, r'_1 + \dots + r'_l, r'_1 + \dots + r'_l; \bar{C}\}:$   
 $\lambda \$ Z_k = Z_k. \checkmark$  [def. 46 (2)]

4.3.2.  $Y_j = r_1 + \dots + r_q, Z_k = r'_1 + \dots + r'_l [; C_k]:$   
 $(r_1 + \dots + r_q) \$ (r'_1 + \dots + r'_l [; C_k])$   
 $= r_1 + \dots + r_q + r'_1 + \dots + r'_l [; C_k]. \checkmark$  [def. 46 (8)]

4.3.3.  $Y_j = r_1 + \dots + r_q; C_j, Z_k = r'_1 + \dots + r'_l [; C_k]:$   
 $(r_1 + \dots + r_q; C_j) \$ (r'_1 + \dots + r'_l [; C_k])$   
 $= r_1 + \dots + r_q + r'_1 + \dots + r'_l; (C_j [; C_k]). \checkmark$  [def. 46 (8)]

5.  $C = C_1 \mathbb{W} C_2$

By induction hypothesis we obtain the following cases for the structure of  $C_1$  and  $C_2$ :

5.1.  $C_1 = \emptyset:$   
 $\emptyset \mathbb{W} C_2 = \emptyset. \checkmark$  [def. 46 (1)]

5.2.  $C_1 = \lambda:$   
 $\lambda \mathbb{W} C_2 = C_2. \checkmark$  [def. 46 (2)]

5.3.  $C_1 = (Y_1 | \dots | Y_o), C_2 = (Z_1 | \dots | Z_p):$   
 $(Y_1 | \dots | Y_o) \mathbb{W} (Z_1 | \dots | Z_p) = |_{j \in [o], k \in [p]} (Y_j \mathbb{W} Z_k)$  [def. 46 (5,6)]

The expression  $\prod_{j \in [o], k \in [p]} (Y_j \mathbb{W} Z_k)$  is in normal form if the  $(Y_j \mathbb{W} Z_k)$  are in normal form.

$$5.3.1. Y_j = \lambda, Z_k \in \{\lambda, r'_1 + \dots + r'_l, r'_1 + \dots + r'_l; \bar{C}\}: \\ \lambda \mathbb{W} Z_k = Z_k. \checkmark \quad [\text{def. 46 (1)}]$$

$$5.3.2. Y_j = r_1 + \dots + r_q, Z_k = r'_1 + \dots + r'_l [; C_k]: \\ (r_1 + \dots + r_q) \mathbb{W} (r'_1 + \dots + r'_l [; C_k]) \\ = r_1 + \dots + r_q + r'_1 + \dots + r'_l [; C_k] \quad [\text{def. 46 (7)}] \\ r_1 + \dots + r_q; (r'_1 + \dots + r'_l [; C_k]) \quad | \\ r'_1 + \dots + r'_l; (r_1 + \dots + r_q [; \mathbb{W} C_k]). \checkmark$$

$$5.3.3. Y_j = r_1 + \dots + r_q; C_j, Z_k = r'_1 + \dots + r'_l [; C_k]: \\ (r_1 + \dots + r_q; C_j) \mathbb{W} (r'_1 + \dots + r'_l [; C_k]) \\ = r_1 + \dots + r_q + r'_1 + \dots + r'_l; (C_j [; \mathbb{W} C_k]) \quad [\text{def. 46 (7)}] \\ r_1 + \dots + r_q; (C_j \mathbb{W} (r'_1 + \dots + r'_l [; C_k])) \quad | \\ r'_1 + \dots + r'_l; ((r_1 + \dots + r_q; C_j) [; \mathbb{W} C_k]). \checkmark \quad \square$$

The proof suggests a recursive procedure to construct the normal form of a parallel expression. The basic parallel expressions  $\emptyset, \lambda$ , and  $x \in X_*$  are already in normal form. Considering a compound expression  $C$ , the respective subexpressions are already in normal form, then we employ the same transformation steps as used in the proof to bring the whole expression  $C$  to normal form, or the subexpressions are not in normal form, then we first have to transform them to normal form.

#### Definition 49. Parallel expression to normal form

Let  $C$  be a parallel expression over a set of rules  $R_*$  and let  $x \in R_*$ . The normal form of  $C$ ,  $NF(C)$ , is recursively constructed as follows.

(Again, we depict the summary of to expressions  $r'_1 + \dots + r'_l$  and  $r'_1 + \dots + r'_l; C_k$  by  $r'_1 + \dots + r'_l [; C_k]$  when both cases can be handled simultaneously.)

Basic cases:

1.  $C = \emptyset$ :  $NF(\emptyset) = \emptyset$ .
2.  $C = \lambda$ :  $NF(\lambda) = \lambda$ .
3.  $C = x$ :  $NF(x) = x$ .

Compound cases:

1.  $C = C_1; C_2$

1.1  $C_1$  already in  $NF$ :

$$1.1.1. C_1 = \emptyset: \\ NF(\emptyset; C_2) = NF(\emptyset) \quad [\text{def. 46 (9)}] \\ = \emptyset. \quad [\text{basic case 1.}]$$



- 1.1.2.  $C_1 = \lambda$ :  
 $NF(\lambda; C_2) = NF(C_2)$ . [def. 46 (12)]
- 1.1.3.  $C_1 = r_1 + \dots + r_m; C'$ :  
 $NF(r_1 + \dots + r_m; C'; C_2) = r_1 + \dots + r_m; C'; C_2$ .
- 1.1.4.  $C_1 = X_1 | \dots | X_n$ :  
 $NF((X_1 | \dots | X_n); C_2) = NF(X_1; C_2 | \dots | X_n; C_2)$ . [def. 46 (14)]  
(We again have to transform  $X_1; C_2 | \dots | X_n; C_2$  to  $NF$  since an  $X_i$  could be  $\lambda$  and  $\lambda; C_2$  would be  $C_2$ , which could be not in  $NF$  )
- 1.2  $C_1$  is not in  $NF$ :  
 $NF(C_1; C_2) = NF(NF(C_1); C_2)$ .
2.  $C = C_1 | C_2$
- 2.1  $C_1$  and  $C_2$  already in  $NF$ :  
 $NF(C_1 | C_2) = C_1 | C_2$ .
- 2.2  $C_1$  is in  $NF$  and  $C_2$  not:  
 $NF(C_1 | C_2) = C_1 | NF(C_2)$ .
- 2.3  $C_1$  is not in  $NF$  and  $C_2$  is in  $NF$ :  
 $NF(C_1 | C_2) = NF(C_1) | C_2$ .
- 2.4  $C_1$  and  $C_2$  are not in  $NF$ :  
 $NF(C_1 | C_2) = NF(C_1) | NF(C_2)$ .
3.  $C = C^*$
- 3.1  $C$  in  $NF$ :
- 3.1.1  $C = \emptyset$ :  
 $NF(\emptyset^*) = NF(\lambda)$  [def. 46 (11)]  
 $= \lambda$ . [basic case 2.]
- 3.1.2  $C = \lambda$ :  
 $NF(\lambda^*) = \lambda$ . [def. 46 (16 b,13)]
- 3.1.3  $C = r_1 + \dots + r_m; C'$ :  
 $NF((r_1 + \dots + r_m; C')^*) = \lambda | (r_1 + \dots + r_m; C'); (r_1 + \dots + r_m; C')^*$ .  
[def. 46 (16)]
- 3.1.4  $C = X_1 | \dots | X_n$ :  
 $NF((X_1 | \dots | X_n)^*) = NF((X'_1 | \dots | X'_m)^*)$  with  $X'_i \neq \lambda, i \in [m]$   
[def. 46 (16 c)]  
 $= NF(\lambda | (X'_1 | \dots | X'_m); (X'_1 | \dots | X'_m)^*)$  [def. 46 (16)]

$$\begin{aligned}
&= NF(\lambda \mid \mid (X'_i; (X'_1 \mid \dots \mid X'_m)^*)) && [\text{def. 46 (14)}] \\
&= \lambda \mid \mid_{i \in [m]} (X'_i; (X'_1 \mid \dots \mid X'_m)^*).
\end{aligned}$$

3.2  $C$  not in  $NF$ :

3.2.1  $C = C'^*$ :  
 $NF((C'^*)^*) = NF(C'^*)$ . [def.46 (16 a)]

3.2.2  $C = (\lambda \mid C')$ :  
 $NF((\lambda \mid C')^*) = NF(C'^*)$ . [def. 46 (16 c)]

3.2.3  $C = (C' \mid \lambda)$ :  
analogous to 3.2.2.

3.2.4 any other cases for the structure of  $C$ :  
 $NF(C^*) = NF(NF(C)^*)$ .

4.  $C = (C_1 \mathbb{W} C_2)$

4.1  $C_1$  and  $C_2$  are in  $NF$ :

4.1.1  $C_1 = \emptyset$ :  $NF(\emptyset \mathbb{W} C_2) = \emptyset$ .

4.1.2  $C_2 = \emptyset$ : analogous to 4.1.1.

4.1.3  $C_1 = \lambda$ :  $NF(\lambda \mathbb{W} C_2) = C_2$ .

4.1.4  $C_2 = \lambda$ : analogous to 4.1.3.

4.1.5  $C_1 = r_1+..+r_o$  and  $C_2 = r'_1+..+r'_p$ :  
 $NF((r_1+..+r_o) \mathbb{W} (r'_1+..+r'_p)) = r_1+..+r_o+r'_1+..+r'_p \mid$   
 $r_1+..+r_o; r'_1+..+r'_p \mid$   
 $r'_1+..+r'_p; r_1+..+r_o$ .

4.1.6  $C_1 = r_1+..+r_o; C'$  and  $C_2 = r'_1+..+r'_p$ :  
 $NF((r_1+..+r_o; C') \mathbb{W} (r'_1+..+r'_p)) = r_1+..+r_o+r'_1+..+r'_p; C' \mid$   
 $r_1+..+r_o; (C' \mathbb{W} (r'_1+..+r'_p)) \mid$   
 $r'_1+..+r'_p; r_1+..+r_o; C'$ .

4.1.7  $C_1 = r_1+..+r_o$  and  $C_2 = r'_1+..+r'_p; C''$ :  
analogous to 4.1.6.

4.1.8  $C_1 = r_1+..+r_o; C'$  and  $C_2 = r'_1+..+r'_p; C''$ :  
 $NF((r_1+..+r_o; C') \mathbb{W} (r'_1+..+r'_p; C'')) = r_1+..+r_o+r'_1+..+r'_p; (C' \mathbb{W} C'') \mid$   
 $r_1+..+r_o; (C' \mathbb{W} (r'_1+..+r'_p; C'')) \mid$   
 $r'_1+..+r'_p; ((r_1+..+r_o; C') \mathbb{W} C'')$ .

$$4.1.9 \quad C_1 = X_1 | \dots | X_n \text{ and } C_2 = r'_1 + \dots + r'_p; C'': \\ NF((X_1 | \dots | X_n) \mathbb{W}(r'_1 + \dots + r'_p; C'')) = \prod_{i \in [n]} NF(X_i \mathbb{W}(r'_1 + \dots + r'_p; C'')).$$

$$4.1.10 \quad C_1 = r_1 + \dots + r_o; C' \text{ and } C_2 = X_1 | \dots | X_m: \\ \text{analogous to 4.1.9.}$$

$$4.1.11 \quad C_1 = X_1 | \dots | X_n \text{ and } C_2 = X'_1 | \dots | X'_m: \\ NF((X_1 | \dots | X_n) \mathbb{W}(X'_1 | \dots | X'_m)) = \prod_{i \in [n], j \in [m]} NF(X_i \mathbb{W} X'_j).$$

$$4.2 \quad C_1 \text{ is in } NF \text{ and } C_2 \text{ not:} \\ NF(C_1 \mathbb{W} C_2) = NF(C_1 \mathbb{W} NF(C_2)).$$

$$4.3 \quad C_1 \text{ is not in } NF \text{ and } C_2 \text{ is:} \\ NF(C_1 \mathbb{W} C_2) = NF(NF(C_1) \mathbb{W} C_2).$$

$$4.4 \quad C_1 \text{ and } C_2 \text{ are not in } NF: \\ NF(C_1 \mathbb{W} C_2) = NF(NF(C_1) \mathbb{W} NF(C_2)).$$

## 5. $C = C_1 \$ C_2$

### 5.1 $C_1$ and $C_2$ are in $NF$ :

$$5.1.1 \quad C_1 = \emptyset: NF(\emptyset \$ C_2) = \emptyset.$$

$$5.1.2 \quad C_2 = \emptyset: \text{analogous to 5.1.1.}$$

$$5.1.3 \quad C_1 = \lambda: NF(\lambda \$ C_2) = C_2.$$

$$5.1.4 \quad C_2 = \lambda: \text{analogous to 5.1.3.}$$

$$5.1.5 \quad C_1 = r_1 + \dots + r_o, C_2 = r'_1 + \dots + r'_p: \\ NF((r_1 + \dots + r_o) \$ (r'_1 + \dots + r'_p)) = r_1 + \dots + r_o + r'_1 + \dots + r'_p.$$

$$5.1.6 \quad C_1 = r_1 + \dots + r_o; C', C_2 = r'_1 + \dots + r'_p: \\ NF((r_1 + \dots + r_o; C') \$ (r'_1 + \dots + r'_p)) = r_1 + \dots + r_o + r'_1 + \dots + r'_p; C'.$$

$$5.1.7 \quad C_1 = r_1 + \dots + r_o, C_2 = r'_1 + \dots + r'_p; C'': \\ \text{analogous to 5.1.6.}$$

$$5.1.8 \quad C_1 = r_1 + \dots + r_o; C', C_2 = r'_1 + \dots + r'_p; C'': \\ NF((r_1 + \dots + r_o; C') \$ (r'_1 + \dots + r'_p; C'')) = r_1 + \dots + r_o + r'_1 + \dots + r'_p; (C' \$ C'').$$

$$5.1.9 \quad C_1 = X_1 | \dots | X_n, C_2 = r'_1 + \dots + r'_p; C'': \\ NF((X_1 | \dots | X_n) \$ (r'_1 + \dots + r'_p; C'')) = \prod_{i \in [n]} NF(X_i \$ (r'_1 + \dots + r'_p; C'')).$$

$$5.1.10 \quad C_1 = r_1 + \dots + r_o; C', C_2 = X_1 | \dots | X_m: \\ \text{analogous to 5.1.9.}$$

$$5.1.11 \quad C_1 = X_1 \mid \dots \mid X_n, \quad C_2 = X'_1 \mid \dots \mid X'_m:$$

$$NF((X_1 \mid \dots \mid X_n)\$(X'_1 \mid \dots \mid X'_m)) = \bigvee_{i \in [n], j \in [m]} NF(X_i \$ X'_j).$$

5.2  $C_1$  is in  $NF$  and  $C_2$  not:

$$NF(C_1 \$ C_2) = NF(C_1 \$ NF(C_2)).$$

5.3  $C_1$  is not in  $NF$  and  $C_2$  is:

$$NF(C_1 \$ C_2) = NF(NF(C_1) \$ C_2).$$

5.4  $C_1$  and  $C_2$  are not in  $NF$ :

$$NF(C_1 \$ C_2) = NF(NF(C_1) \$ NF(C_2)).$$

The following example demonstrates the construction of the normal form for a parallel expression.

**Example 5.** Constructing the normal form of a parallel expression

Let  $(r_1; r_2; r_3)\$(r_4; r_5)^*; r_6; r_7 \$ r_8$  be a parallel expression. Its normal form can be constructed as follows according to definition 49. Since all transformation steps refer to the same definition we only reference the respective parts. In order to make clear which parts of the expression are modified in the next construction step we underline the respective parts.

$$\begin{aligned}
& NF(\underline{(r_1; r_2; r_3)}\$(\underline{(r_4; r_5)^*}; r_6; r_7 \$ r_8)) \\
&= NF(\underline{(r_1; r_2; r_3)} \$ NF(\underline{(r_4; r_5)^*}; r_6; r_7 \$ r_8)) \quad [5.2] \\
&= NF(\underline{(r_1; r_2; r_3)} \$ NF(NF(\underline{(r_4; r_5)^*}); r_6; r_7 \$ r_8)) \quad [1.2] \\
&= NF(\underline{(r_1; r_2; r_3)} \$ NF(\underline{(\lambda \mid r_4; r_5; (r_4; r_5)^*)}; r_6; r_7 \$ r_8)) \quad [3.1.3] \\
&= NF(\underline{(r_1; r_2; r_3)} \$ NF(\underline{(\lambda; r_6; r_7 \$ r_8)} \mid \underline{(r_4; r_5; (r_4; r_5)^*}; r_6; r_7 \$ r_8))) \quad [1.1.1] \\
&= NF(\underline{(r_1; r_2; r_3)} \$ (NF(\underline{\lambda; r_6; r_7 \$ r_8}) \mid \underline{(r_4; r_5; (r_4; r_5)^*}; r_6; r_7 \$ r_8))) \quad [2.3] \\
&= NF(\underline{(r_1; r_2; r_3)} \$ (\underline{(r_6; r_7 \$ r_8)} \mid \underline{(r_4; r_5; (r_4; r_5)^*}; r_6; r_7 \$ r_8))) \quad [1.1.2] \\
&= NF(\underline{(r_1; r_2; r_3)} \$ (r_6; r_7 \$ r_8) \mid \underline{(r_1; r_2; r_3)} \$ (r_4; r_5; (r_4; r_5)^*; r_6; r_7 \$ r_8)) \quad [5.1.10] \\
&= NF(\underline{(r_1; r_2; r_3)} \$ (r_6; r_7 \$ r_8)) \mid NF(\underline{(r_1; r_2; r_3)} \$ (r_4; r_5; (r_4; r_5)^*; r_6; r_7 \$ r_8)) \quad [2.4] \\
&= r_{1+r_6}; ((r_2; r_3)\$(r_7 \$ r_8)) \mid r_{1+r_4}; ((r_2; r_3)\$(r_5; (r_4; r_5)^*; r_6; r_7 \$ r_8)) \quad [5.1.8]
\end{aligned}$$

The normal form of  $(r_1; r_2; r_3)\$(r_4; r_5)^*; r_6; r_7 \$ r_8$  provides two choices:  $r_{1+r_6}; ((r_2; r_3)\$(r_7 \$ r_8))$  with  $r_{1+r_6}$  is the first parallel rule to be applied and  $r_{1+r_4}; ((r_2; r_3)\$(r_5; (r_4; r_5)^*; r_6; r_7 \$ r_8))$  with  $r_{1+r_4}$  as first parallel rule.

### Semantics of a parallel expression employing the normal form

The semantics of a parallel expression employing the normal form builds permitted derivations stepwise. Starting from the normal form of the parallel expression the first derivation step is performed with the provided parallel

rule. Then the remaining expression again is transformed to normal form and the next derivation step is performed. This process goes on until the whole expression is dissolved or none of the provided parallel rules could be applied. In case the parallel expression containing some subexpression  $C^*$  the derivation process could go on infinitely, but frequently will provide permitted derivations.

The following definition introduces this procedure formally.

**Definition 50.**  $SEM_{II}$

Let  $C$  be a parallel expression. The semantics of the second approach employing the normal form of  $C$  is defined by:

$SEM_{II}(C) = SM_{NF}(NF(C))$  with

1.  $SM_{NF}(\emptyset) = \emptyset$ ,
2.  $SM_{NF}(X_1 \mid \dots \mid X_n) = SM_{NF}(X_1) \cup \dots \cup SM_{NF}(X_n)$ ,
3.  $SM_{NF}(\lambda) = \{G \xrightarrow{0} G \mid G \in \mathcal{G}\}$ ,
4.  $SM_{NF}(r) = \{G \xrightarrow[r]{} G' \mid G, G' \in \mathcal{G}\}$ ,
5.  $SM_{NF}(r; C) = \{G \xrightarrow[r]{} G' \mid G, G' \in \mathcal{G}\} \circ SM_{NF}(NF(C))$ .

In the following both semantic approaches are illustrated by respectively one example.

**Example 6.** *count-down*

A typical example of synchronised parallelism is a step counter that combines a graph transformation unit with a counting mechanism. A very simple counting mechanism is implemented by the transformation unit *count-down* which gets a node with a number of loops as input and has a rule that removes a loop if applicable.

*count-down*

$$\begin{aligned} \text{initial} &: gr(\mathbb{N}) \\ \text{rules} : tic &= \bullet \curvearrowright \supseteq \bullet \subseteq \bullet \end{aligned}$$

where  $gr(\mathbb{N})$  contains all graphs  $gr(n) = (\{\bullet\}, [n + 1], s, t, l)$  with  $s(i) = t(i) = \bullet$  and  $l(i) = *$  for  $i \in [n + 1]$ .

Let  $gtu = (I, P, C, T)$  be a graph transformation unit with the rules  $r_1, \dots, r_k$

such that the label  $*$  is not used in  $gtu$ . Then  $gtu$  can be combined with the *count-down* unit yielding a step counter for  $gtu$ :

```

stepcount(gtu)
  import : gtu, count-down
  initial :  $I_{gtu} + gr(\mathbb{N})$ 
  rules :  $r_1, \dots, r_k, tic$ 
  control :  $((r_1 \mid \dots \mid r_k)\$tic)^*$ 
  terminal :  $T_{gtu} + gr(\mathbb{N})$ 

```

Starting with  $G + gr(n)$  for  $G \in SEM(I)$  and  $n \in \mathbb{N}$ , the control condition requires that the rules of  $gtu$  must run synchronously with the  $tic$  rule. But if such a run takes more than  $n$  steps, then all unlabelled loops are removed such that there is no longer a subgraph from  $gr(\mathbb{N})$  and no terminal graph can be reached. As also the control condition  $C$  must hold for the  $gtu$ -part of the runs, the semantics of  $stepcount(gt_u)$  restricts the semantics of  $gtu$  to the derivations that are not longer than the given  $n$ .

For this example we employ the language control. Hence, we first construct the language of the control condition  $((r_1 \mid \dots \mid r_k)\$tic)^*$ .

$$\begin{aligned}
L(((r_1 \mid \dots \mid r_k)\$tic)^*) &= L((r_1 \mid \dots \mid r_k)\$tic)^* && [\text{def. 41(6)}] \\
&= (L(r_1 \mid \dots \mid r_k)\$L(tic))^* && [\text{def. 41(7)}] \\
&= ((L(r_1) \cup \dots \cup L(r_k))\$L(tic))^* && [\text{def. 41(5)}] \\
&= (\{r_1, \dots, r_k\}\$\{tic\})^* && [\text{def. 41(3)}] \\
&= \{r_1\$tic, \dots, r_k\$tic\}^* && [\text{def. 40}] \\
&= \{r_1+tic, \dots, r_k+tic\}^* && [\text{def. 39, exa. 2}]
\end{aligned}$$

Now we could build all possible derivations with rule application sequences provided by  $\{r_1+tic, \dots, r_k+tic\}^*$ .

$$SEM(stepcount(gt_u)) = \{G_0 \xrightarrow{\bar{r}_1} G_1 \xrightarrow{\bar{r}_2} \dots \xrightarrow{\bar{r}_n} G_n \mid \bar{r}_1 \bar{r}_2 \dots \bar{r}_n \in \{r_1+tic, \dots, r_k+tic\}^*, G_0, \dots, G_n \in \mathcal{G}\}$$

### Example 7. *Quidditch*

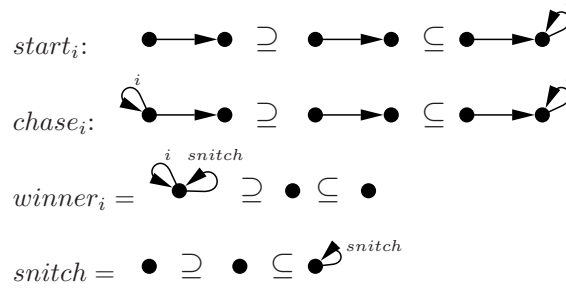
This example presents the transformation unit *Quidditch*, which models two players (seekers) chasing after a ball, the snitch. The player who reaches the snitch first wins.

*Quidditch*

```

initial : undirected & unlabelled & loopfree
rules :

```



$control : snatch; start_1+start_2; chase_1^* \$ chase_2^*; (winner_1 \mid winner_2)$

*Quidditch* runs on an undirected, unlabelled, and loop-free graph. First, *snatch* is applied at a node. Secondly, the two seekers choose a target of an edge as start node and then, according to the control condition, they chase synchronously along edges. The seeker who happens to reach the node of the *snatch* can be the winner (if the respective rule is applied). As the start nodes are targets of edges and as the graph is undirected, the *chase* rules can always be applied from the start on.

For the *Quidditch* example we employ the normal form control. According to Definition 50 the semantics of the units control condition is given by  $SM_{NF}(NF(snatch; start_1+start_2; (chase_1 \$ chase_2)^*; (winner_1 \mid winner_2)))$  and evaluated as follows. (As we are only interested to see which rules are applied in parallel and in which order we use no actual graphs but dummy graphs  $G, G'$ , which stand for any suitable graphs.)

We abbreviate the rule names in order to keep the lines shorter

$$\begin{aligned}
& SM_{NF}(NF(sn; st_1+st_2; (chs_1 \$ chs_2)^*; (win_1 \mid win_2))) \\
&= SM_{NF}(sn; st_1+st_2; (chs_1 \$ chs_2)^*; (win_1 \mid win_2)) \quad [49(1.1.3)] \\
&= \{G \Rightarrow G'\}_{sn} \circ SM_{NF}(NF(st_1+st_2; (chs_1 \$ chs_2)^*; (win_1 \mid win_2))) \\
&= \{G \Rightarrow G'\}_{sn} \circ SM_{NF}(st_1+st_2; (chs_1 \$ chs_2)^*; (win_1 \mid win_2)) \quad [49(1.1.3)] \\
&= \{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ SM_{NF}(NF((chs_1 \$ chs_2)^*; (win_1 \mid win_2))) \quad [50(5)] \\
&= \{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ SM_{NF}(NF(NF((chs_1 \$ chs_2)^*; (win_1 \mid win_2))) \quad [49(1.2)] \\
&= \{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ SM_{NF}(NF(NF(NF(chs_1 \$ chs_2)^*; (win_1 \mid win_2))) \quad [49(3.2.4)] \\
&= \{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ SM_{NF}(NF(NF((chs_1+chs_2)^*; (win_1 \mid win_2))) \quad [49(5.1.5)]
\end{aligned}$$

Since  $\{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2}$  is a completed derivation pattern and does not alter anymore we replace it by  $D$  in order to keep the example clear

$$\begin{aligned}
&= D \circ SM_{NF}(NF((\lambda \mid (ch_1+chs_2); (chs_1+chs_2)^*; (win_1 \mid win_2)))) \quad [49(3.1.3)] \\
&= D \circ SM_{NF}(NF(\lambda; (win_1 \mid win_2) \mid (chs_1+chs_2); (chs_1+chs_2)^*; (win_1 \mid win_2))) \quad [49(1.1.4)] \\
&= D \circ SM_{NF}(NF((win_1 \mid win_2) \mid (chs_1+chs_2); (chs_1+chs_2)^*; (win_1 \mid win_2)))
\end{aligned}$$

$$\begin{aligned}
&= D \circ \mathcal{SM}_{NF}((win_1|win_2) \mid (chs_1+chs_2); (chs_1+chs_2)^*; (win_1|win_2))) \quad [49(2.1)] \\
&= D \circ (\mathcal{SM}_{NF}(win_1) \cup \mathcal{SM}_{NF}(win_2) \cup \mathcal{SM}_{NF}((chs_1+chs_2); (chs_1+chs_2)^*; (win_1|win_2))) \\
&\quad [50(2)] \\
&= D \circ (\{G \Rightarrow G'\}_{win_1} \cup \{G \Rightarrow G'\}_{win_2} \cup \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \mathcal{SM}_{NF}(NF((ch_1+ch_2)^*; (win_1 \mid win_2)))) \\
&\quad [50(4,5)] \\
&= D \circ \{G \Rightarrow G'\}_{win_1} \cup D \circ \{G \Rightarrow G'\}_{win_2} \cup D \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \mathcal{SM}_{NF}(NF((ch_1+ch_2)^*; (win_1 \mid win_2))) \\
&\quad D \circ \{G \Rightarrow G'\}_{win_1} \text{ and } D \circ \{G \Rightarrow G'\}_{win_2} \text{ are completed derivation patterns. Therefore we} \\
&\quad \text{omit them at the moment. We continue the evaluation with } D \circ \{G \Rightarrow G'\}_{ch_1+chs_2} \circ \\
&\quad \mathcal{SM}_{NF}(NF((chs_1+chs_2)^*; (win_1 \mid win_2))) \text{ which strongly reminds of the expres-} \\
&\quad \text{sion in line four of the evaluation.} \\
&\quad D \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \mathcal{SM}_{NF}(NF((chs_1+chs_2)^*; (win_1|win_2))) \\
&= \dots \\
&= D \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ (\{G \Rightarrow G'\}_{win_1} \cup \{G \Rightarrow G'\}_{win_2} \cup \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \mathcal{SM}_{NF}(NF((chs_1+chs_2)^*; (win_1|win_2)))) \\
&= D \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \{G \Rightarrow G'\}_{win_1} \cup \\
&\quad D \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \{G \Rightarrow G'\}_{win_2} \cup \\
&\quad D \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \mathcal{SM}_{NF}(NF((chs_1+chs_2)^*; (win_1 \mid win_2)))
\end{aligned}$$

At this point we cut of the evaluation since it should be clear how it works. We yielded the completed derivation patterns

$$\begin{aligned}
&\{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ \{G \Rightarrow G'\}_{win_1}, \\
&\{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ \{G \Rightarrow G'\}_{win_2}, \\
&\{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \{G \Rightarrow G'\}_{win_1}, \text{ and} \\
&\{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \{G \Rightarrow G'\}_{win_2}.
\end{aligned}$$

The first two representing one seeker finds the snitch without chasing it (i.e. the snitch was at its start node) and the other two representing both seekers synchronously make a chase step and then one finds the snitch. Moreover, we yielded the non complete pattern  $\{G \Rightarrow G'\}_{sn} \circ \{G \Rightarrow G'\}_{st_1+st_2} \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \{G \Rightarrow G'\}_{ch_1+ch_2} \circ \mathcal{SM}_{NF}(NF((chs_1+chs_2)^*; (win_1|win_2)))$  which has to be evaluated further. As one can expect each evaluation of  $\mathcal{SM}_{NF}(NF((chs_1+chs_2)^*; (win_1 \mid win_2)))$  yields  $\{G \Rightarrow G'\}_{win_1}$  and  $\{G \Rightarrow G'\}_{win_2}$  completing the derivation pattern or provides another synchronous chase step  $\{G \Rightarrow G'\}_{ch_1+ch_2}$  again followed by  $\mathcal{SM}_{NF}(NF((chs_1+chs_2)^*; (win_1 \mid win_2)))$ .



# Chapter 4

## Languages of Parallel Expressions are Regular

The last chapter has introduced parallel expressions as an extension of regular expressions by respectively one parallel operator. Moreover, it has shown that parallel expressions describe languages. This chapter now shows that the languages of parallel expressions are still regular despite the additional parallel operators.

**Proposition 51.** Let  $C$  be a parallel expression.  
The language  $L(C)$  is regular.

To prove this proposition we use the fact that finite state automata recognise the class of regular languages (cf., e.g. [Ric08]). I.e. if we are able to build finite state automata from parallel expressions, which recognise the languages of these expressions, we have a proof that the language of a parallel expression is regular. Therefore in a first step we construct finite state automata from parallel expressions, and then show that the language of these 'parallel automata' is equal to the language of their respective parallel expression.

### 4.1 Parallel Expressions to Automata

The construction of finite state automata from parallel expressions bases on the recursive automata construction from regular expressions and adds automata constructions for weakly parallel and synchronously composed expressions.

The automata constructions for regular expressions as we assume them are given in the preliminaries.

**Definition 52. Constructing automata from parallel expressions**

Let  $C_1, C_2, C$  be parallel expressions over  $X_*$  and  $r \in X_*$ .

- $A(\emptyset), A(\lambda), A(r), A(C_1; C_2), A(C_1|C_2), A(C^*)$  are constructed as for regular expressions,
- $A(C_1 \parallel C_2) = A(C_1) \parallel A(C_2)$ ,
- $A(C_1 \$ C_2) = A(C_1) \$ A(C_2)$ .

The automata constructions for parallel composed expressions employ the weak parallel respectively synchronous composition of two automata, also denoted by  $\parallel$  and  $\$$ , which are introduced in the following.

**Parallel composition of automata**

The parallel composition of two automata should result in an automaton which executes both input automata simultaneously according to the given parallel operator. I.e. in every execution step the composition automaton should provide a transition labeled with  $r_1+r_2$  whenever the input automata in this step provide transitions label with  $r_1$  respectively  $r_2$ . Moreover, the weak parallel composition automaton should provide two more transitions one labeled with  $r_1$  and one with  $r_2$ , whereas the synchronous composition automaton should only provide a transition representing one of the input automata operating on its own, when the other has reached a final state.

As a basis for our parallel automata compositions, we make use of the product automaton, a concept from automata theory. The product automaton combines two input automata in such a way that it makes a transition labeled with  $x$  whenever the two input automata make a transition with  $x$ . To achieve this behaviour the states of the product automaton are comprised of state pairs, each component representing one of the input automata. The product automaton then makes a transition label with  $x$  from one state pair to another if both input automata do so regarding their respective states and state transitions. A formal definition of a product automata is given in the preliminaries.

In order to realise the weakly parallel and synchronous composition of two automata we have to adapt this construction in order to meet our requirements. In the following we first define the weakly parallel and afterwards the synchronous composition of two automata.

### Weak parallel composition of automata

The weak parallel composition of two automata  $A_1$  and  $A_2$  combines the state sets  $S_1$  and  $S_2$  of the input automata to pairs of states  $S_1 \times S_2$ . The input alphabet comprises the set of all parallel rules over  $I_1$  and  $I_2$ ,  $I_1| \times |I_2$ , as well as  $I_1$  and  $I_2$  on their own. It allows transitions from a state pair  $(s_1, s_2)$  labeled with  $r_1+r_2$  to a pair  $(s'_1, s'_2)$  if there are respective transitions in  $A_1$  and  $A_2$  as well as transitions labeled with only a single rule to a pair  $(s'_1, s_2)$  (respectively  $(s_1, s'_2)$ ) reflecting that only one input automaton makes a transition and the other waits.

#### Definition 53. Weak parallel composition of finite state automata

Let  $A_i = (S_i, I_i, d_i, s_{0_i}, F_i), i \in \{1, 2\}$  be two finite state automata. The weak parallel composition of  $A_1$  and  $A_2$  is defined by

$$\begin{aligned} A_1 \parallel A_2 &= (S, I, d_w, (s_{0_1}, s_{0_2}), F) \text{ with} \\ S &= S_1 \times S_2, \\ I &= (I_1| \times |I_2) \cup I_1 \cup I_2, \\ d_w &= \{((s_1, s_2), r_1+r_2, (s'_1, s'_2)) \mid (s_1, r_1, s'_1) \in d_1, (s_2, r_2, s'_2) \in d_2\} \cup \\ &\quad \{((s_1, s_2), r_1, (s'_1, s_2)) \mid (s_1, r_1, s'_1) \in d_1\} \cup \\ &\quad \{((s_1, s_2), r_2, (s_1, s'_2)) \mid (s_2, r_2, s'_2) \in d_2\} \\ F &= F_1 \times F_2. \end{aligned}$$

To illustrate the weak parallel composition, Figure 4.1 depicts two automata and their weak parallel composition.

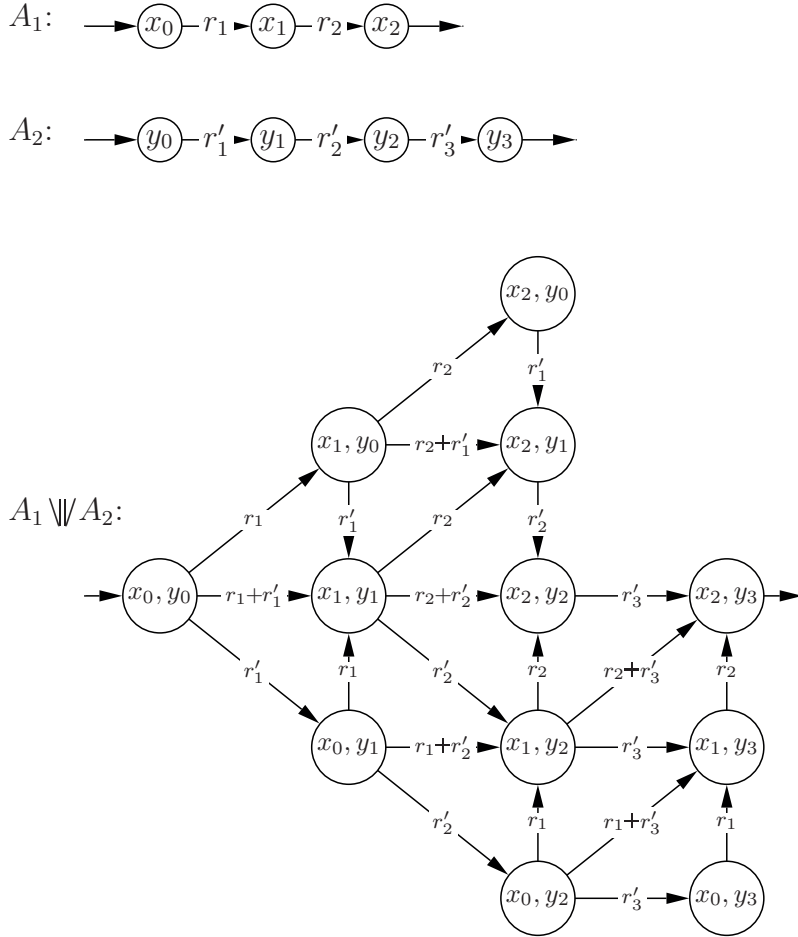


Figure 4.1: Weak parallel composition of two automata  $A_1$  and  $A_2$

### Synchronous composition of automata

The synchronous composition of two automata resembles the weak parallel composition except for that individual transitions of one input automaton are only allowed if the other has reached a final state. In such a case it has to be ensured that the latter automaton makes no more transitions. For that reason the states of the synchronous composition automaton not only contain state pairs, but also the single states of the input automata. If one of the input automata has come to an end the corresponding transition of the composition automaton leads from a state pair to a single state of the automata which operates alone. So it is ensured that the automaton which has reached a final state can make no more transitions after it stops operating.

**Definition 54. Synchronous composition of finite state automata**

Let  $A_i = (S_i, I_i, d_i, s_{0_i}, F_i), i \in \{1, 2\}$  be two finite state automata. The synchronous composition of  $A_1$  and  $A_2$  is defined by

$$A_1 \$ A_2 = (S, I, d_{\#}, (s_{0_1}, s_{0_2}), F)$$

$$S = (S_1 \times S_2) \cup S_1 \cup S_2,$$

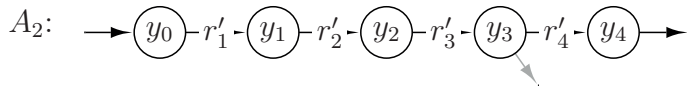
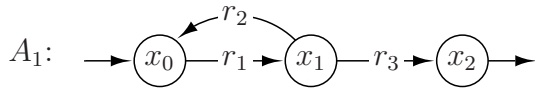
$$I = (I_1 \times I_2) \cup I_1 \cup I_2,$$

$$d_{\#} = \{((s_1, s_2), r_1+r_2, (s'_1, s'_2)) \mid (s_1, r_1, s'_1) \in d_1, (s_2, r_2, s'_2) \in d_2\} \cup \\ \{((s_1, s_2), r_1, s'_1) \mid (s_1, r_1, s'_1) \in d_1, s_2 \in F_2\} \cup \\ \{((s_1, s_2), r_2, s'_2) \mid s_1 \in F_1, (s_2, r_2, s'_2) \in d_2\} \cup \\ d_1 \cup d_2,$$

$$F = (F_1 \times F_2) \cup F_1 \cup F_2.$$

Figure 4.2 depicts the synchronous composition of two automata  $A_1$  and  $A_2$  omitting unreachable states.

Automata  $A_1$  and  $A_2$ :



Synchronous composition  $A_1 \$ A_2$ :

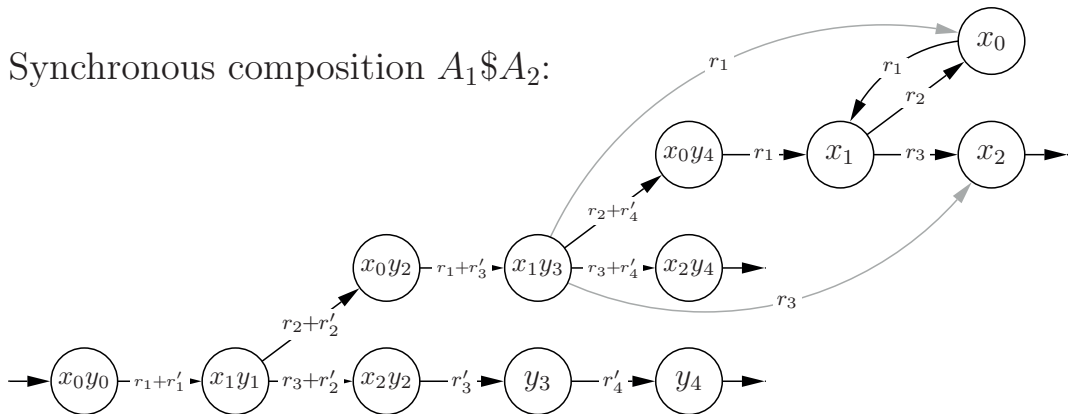


Figure 4.2: Synchronous composition of two automata  $A_1$  and  $A_2$

## 4.2 Automata Recognise Parallel Languages

Now, after we have defined automata for parallel expressions we have to prove that these automata actually recognise the language of parallel expressions, i.e.  $L(A(C)) = L(C)$  for a parallel expression  $C$ . This is done by induction over the structure of parallel expressions. Since finite state automata recognise regular languages this proof is already done for the induction begin and induction steps for the possible structures  $C_1; C_2$ ,  $C_1|C_2$ , and  $C^*$  (see, e.g., [Ric08]). What is left to do are the induction steps for the structures  $C_1 \mathbb{W} C_2$  and  $C_1 \$ C_2$ , i.e. the proofs for  $L(A(C_1 \mathbb{W} C_2)) = L(C_1 \mathbb{W} C_2)$  and  $L(A(C_1 \$ C_2)) = L(C_1 \$ C_2)$ , provided that  $L(A(C_1)) = L(C_1)$  and  $L(A(C_2)) = L(C_2)$ .

### Weak parallel case

**Proposition 55.** Induction step for  $C_1 \mathbb{W} C_2$

Let  $C_1$  and  $C_2$  be two parallel expressions, and let  $L(A(C_1)) = L(C_1)$  and  $L(A(C_2)) = L(C_2)$ . Then it holds

$$L(A(C_1 \mathbb{W} C_2)) = L(C_1 \mathbb{W} C_2).$$

*Proof.* Let  $A(C_i) = (S_i, I_i, d_i, s_{0_i}, F_i), i \in \{1, 2\}$  be the two finite state automata for  $C_1$  and  $C_2$ , and let  $A(C_1) \mathbb{W} A(C_2) = (S, I, d_w, (s_{0_1}, s_{0_2}), F)$  the weak parallel composition of  $A(C_1)$  and  $A(C_2)$ .

$$\begin{aligned}
L(A(C_1 \mathbb{W} C_2)) &= L(A(C_1) \mathbb{W} A(C_2)) && \text{[def. 52 (A(C))]} \\
&= \{w \mid (s'_1, s'_2) \in d_w^*((s_{0_1}, s_{0_2}), w), && \text{[def. L(A)]} \\
&\quad (s'_1, s'_2) \in F\} \\
&= \{w \mid w \in w_1 \mathbb{W} w_2, && \text{[lemma 56]} \\
&\quad s'_1 \in d_1^*(s_{0_1}, w_1), \\
&\quad s'_2 \in d_2^*(s_{0_2}, w_2), \\
&\quad (s'_1, s'_2) \in F\} \\
&= \{w \mid w \in w_1 \mathbb{W} w_2, && \text{[def. 53 (A}_1 \mathbb{W} A_2)] \\
&\quad s'_1 \in d_1^*(s_{0_1}, w_1), \\
&\quad s'_2 \in d_2^*(s_{0_2}, w_2), \\
&\quad s'_1 \in F_1, s'_2 \in F_2\} \\
&= \{w \mid w \in w_1 \mathbb{W} w_2, && \text{[def. L(A)]} \\
&\quad w_1 \in L(A(C_1)), \\
&\quad w_2 \in L(A(C_2))\} \\
&= \bigcup (w_1 \mathbb{W} w_2) \text{ with } w_1 \in L(A(C_1)), w_2 \in L(A(C_2)) \\
&= L(A(C_1)) \mathbb{W} L(A(C_2)) && \text{[def. 40 (L}_1 \mathbb{W} L_2)]
\end{aligned}$$

$$\begin{aligned}
&= L(C_1) \mathbb{W} L(C_2) && \text{[i.h.]} \\
&= L(C_1 \mathbb{W} C_2). && \text{[def. 41 (L(C))]} \quad \square
\end{aligned}$$

Lemma 56 is the centerpiece of the preceding proof. It states that whenever a weak parallel composition automata is able to process a word  $w$  its input automata are able to process words  $w_1$  respectively  $w_2$  and  $w$  is a weak parallel composition of  $w_1$  and  $w_2$ .

**Lemma 56.** Let  $A_i = (S_i, I_i, d_i, s_{i_0}, F_i), i \in \{1, 2\}$  be two finite state automata and  $A = (S, I, d_w, (s_{0_1}, s_{0_2}), F)$  be the weak parallel composition  $A_1 \mathbb{W} A_2$ . Then for all  $w \in I_*^*$  holds:

$$(s'_1, s'_2) \in d_w^*((s_1, s_2), w) \Leftrightarrow s'_1 \in d_1^*(s_1, w_1), s'_2 \in d_2^*(s_2, w_2), w \in w_1 \mathbb{W} w_2.$$

*Proof.*

Induction over the structure of  $w$ .

*Basis:*  $w = \lambda$ :

$$\begin{aligned}
(s'_1, s'_2) \in d_w^*((s_1, s_2), \lambda) &\Leftrightarrow s'_1 = s_1, s'_2 = s_2 \\
&\Leftrightarrow s'_1 \in d_1^*(s_1, \lambda), s'_2 \in d_2^*(s_2, \lambda), \lambda \in \lambda \mathbb{W} \lambda.
\end{aligned}$$

*Hypothesis:* For a  $w \in I_*^*$  holds:

$$(s'_1, s'_2) \in d_w^*((s_1, s_2), w) \Leftrightarrow s'_1 \in d_1^*(s_1, w_1), s'_2 \in d_2^*(s_2, w_2), w \in w_1 \mathbb{W} w_2.$$

*Step:* Proof for  $wx$  with  $x \in I_*$ .

$$\begin{aligned}
(s'_1, s'_2) \in d_w^*((s_1, s_2), wx) &\Leftrightarrow (s'_1, s'_2) \in d_w((\bar{s}_1, \bar{s}_2), x) \text{ for some} && \text{[def. } d^*] \\
&\quad (\bar{s}_1, \bar{s}_2) \in d_w^*((s_1, s_2), w) \\
&\Leftrightarrow (s'_1, s'_2) \in d_w((\bar{s}_1, \bar{s}_2), x) \text{ for some} && \text{[i.h.]} \\
&\quad \bar{s}_1 \in d_1^*(s_1, w_1), \bar{s}_2 \in d_2^*(s_2, w_2), w \in w_1 \mathbb{W} w_2
\end{aligned}$$

According to the definition of  $d_w$  we obtain three possibilities how  $d_w((\bar{s}_1, \bar{s}_2), x)$  could be obtained from  $d_1$  and  $d_2$

$$\begin{aligned}
&\Leftrightarrow (s'_1, s'_2) \in \{(\ddot{s}_1, \ddot{s}_2) \mid (\ddot{s}_1 \in d_1(\bar{s}_1, x_1), \ddot{s}_2 \in d_2(\bar{s}_2, x_2), x = x_1 + x_2) && \text{[def. 53]} \\
&\quad \text{or} \\
&\quad (\ddot{s}_1 \in d_1(\bar{s}_1, x), \ddot{s}_2 = \bar{s}_2) \\
&\quad \text{or} \\
&\quad (\ddot{s}_1 = \bar{s}_1, \ddot{s}_2 \in d_2(\bar{s}_2, x))\} \text{ for some} \\
&\quad \bar{s}_1 \in d_1^*(s_1, w_1), \bar{s}_2 \in d_2^*(s_2, w_2), w \in w_1 \mathbb{W} w_2
\end{aligned}$$

Now concerning  $d_1$  and  $d_2$  we again use the definition of  $d^*$  (but backwards) in order to combine the single transition steps for  $x_1$  and  $x_2$  resp  $x$  with the respective state transitions  $d_1^*(s_1, w_1)$  and  $d_2^*(s_2, w_2)$ .

$$\Leftrightarrow (s'_1, s'_2) \in \{(\ddot{s}_1, \ddot{s}_2) \mid (\ddot{s}_1 \in d_1^*(s_1, w_1 x_1), \ddot{s}_2 \in d_2^*(s_2, w_2 x_2), x = x_1 + x_2) && \text{[def. } d^*]$$

or

$$\begin{aligned} \ddot{s}_1 \in d_1^*(s_1, w_1x), \ddot{s}_2 = \bar{s}_2 \in d_2^*(s_2, w_2) \\ \text{or} \\ \ddot{s}_1 = \bar{s}_1 \in d_1^*(s_1, w_1), \ddot{s}_2 \in d_2^*(s_2, w_2x), \\ w \in w_1 \mathbb{W}w_2 \end{aligned}$$

Since we want to prove that  $wx$  is a weakly composition of a word processed by  $A_1$  and a word processed by  $A_2$  we add the information we have so far about  $wx$  leaving the information how the state pair  $(s'_1, s'_2)$  arises from  $d_1^*$  and  $d_2^*$ , which we do not need anymore.

$$\Leftrightarrow s'_1 \in d_1^*(s_1, w_1x_1), s'_2 \in d_2^*(s_2, w_2x_2), wx \in (w_1 \mathbb{W}w_2)\{x_1+x_2\}, w \in w_1 \mathbb{W}w_2$$

or

$$s'_1 \in d_1^*(s_1, w_1x), s'_2 \in d_2^*(s_2, w_2), wx \in (w_1 \mathbb{W}w_2)\{x\}, w \in w_1 \mathbb{W}w_2$$

or

$$s'_1 \in d_1^*(s_1, w_1), s'_2 \in d_2^*(s_2, w_2x), wx \in (w_1 \mathbb{W}w_2)\{x\}, w \in w_1 \mathbb{W}w_2$$

Lemma 57, which is given subsequently to the proof, allows to infer that in every case  $wx$  is contained in the weakly composition of the word processed by  $d_1^*$  and the word processed by  $d_2^*$ .

$$\Leftrightarrow s'_1 \in d_1^*(s_1, w_1x_1), s'_2 \in d_2^*(s_2, w_2x_2), wx \in (w_1x_1) \mathbb{W}(w_2x_2) \quad [\text{lemma 57}]$$

or

$$s'_1 \in d_1^*(s_1, w_1x), s'_2 \in d_2^*(s_2, w_2), wx \in (w_1x) \mathbb{W}w_2$$

or

$$s'_1 \in d_1^*(s_1, w_1), s'_2 \in d_2^*(s_2, w_2x), wx \in w_1 \mathbb{W}(w_2x). \quad \square$$

**Lemma 57.** Let  $w, w_1, w_2 \in \mathcal{R}_*$ ,  $x, x_1, x_2 \in \mathcal{R}_*$  and  $w \in w_1 \mathbb{W}w_2$ . Then the following holds:

- a)  $wx \in (w_1 \mathbb{W}w_2)\{x_1 + x_2\} \Leftrightarrow wx \in (w_1x_1) \mathbb{W}(w_2x_2)$ ,
- b)  $wx_1 \in (w_1 \mathbb{W}w_2)\{x_1\} \Leftrightarrow wx_1 \in (w_1x_1) \mathbb{W}w_2$ ,
- c)  $wx_2 \in (w_1 \mathbb{W}w_2)\{x_2\} \Leftrightarrow wx_2 \in w_1 \mathbb{W}(w_2x_2)$ .

*Proof.*

a)

' $\Rightarrow$ ':

$$\begin{aligned} wx \in (w_1 \mathbb{W}w_2)\{x_1 + x_2\} &\Rightarrow wx \in ((w_1 \mathbb{W}w_2)\{x_1 + x_2\} \cup \\ &\quad (w_1 \mathbb{W}(w_2x_2))\{x_1\} \cup \\ &\quad ((w_1x_1) \mathbb{W}w_2)\{x_2\}) \quad [\text{def. } \cup, \in] \\ &\Leftrightarrow wx \in (w_1x_1) \mathbb{W}(w_2x_2). \quad [\text{lemma 58}] \end{aligned}$$

' $\Leftarrow$ ':

$$\begin{aligned} wx \in (w_1x_1) \mathbb{W}(w_2x_2) &\Leftrightarrow wx \in ((w_1 \mathbb{W}w_2)\{x_1 + x_2\} \cup \\ &\quad (w_1 \mathbb{W}(w_2x_2))\{x_1\} \cup \\ &\quad ((w_1x_1) \mathbb{W}w_2)\{x_2\}) \quad [\text{lemma 58}] \\ &\Rightarrow wx \in (w_1 \mathbb{W}w_2)\{x_1 + x_2\}. \quad [w \in w_1 \mathbb{W}w_2] \end{aligned}$$

b)



$$\begin{aligned}
&\text{case 1: } w_2 = \lambda \text{ (contains } w_1 = \lambda \text{ and } w_1 \neq \lambda\text{):} \\
wx_1 \in (w_1 \mathbb{W} \lambda) \{x_1\} &\Leftrightarrow wx_1 \in \{w_1\} \{x_1\} && [\text{def. 39}] \\
&\Leftrightarrow wx_1 \in \{w_1 x_1\} \\
&\Leftrightarrow wx_1 \in (w_1 x_1) \mathbb{W} \lambda. && [\text{def. 39}]
\end{aligned}$$

$$\begin{aligned}
&\text{case 2: } w_2 = \bar{w}_2 b, b \in \mathcal{R}_* \text{ (contains } w_1 = \lambda \text{ and } w_1 \neq \lambda\text{):} \\
wx_1 \in (w_1 \mathbb{W} (\bar{w}_2 b)) \{x_1\} &\Leftrightarrow wx_1 \in (w_1 \mathbb{W} (\bar{w}_2 b)) \{x_1\} \cup && [\text{def. } \cup \text{ and } (*)] \\
&\quad (w_1 \mathbb{W} \bar{w}_2) \{x_1 + b\} \cup \\
&\quad ((w_1 x_1) \mathbb{W} \bar{w}_2) \{b\} \\
&\Leftrightarrow wx_1 \in ((w_1 x_1) \mathbb{W} (\bar{w}_2 b)). && [\text{lemma 58}]
\end{aligned}$$

c)

$$\begin{aligned}
&\text{case 1: } w_1 = \lambda \text{ (contains } w_2 = \lambda \text{ and } w_2 \neq \lambda\text{):} \\
wx_2 \in (\lambda \mathbb{W} w_2) \{x_2\} &\Leftrightarrow wx_2 \in \{w_2\} \{x_2\} \\
&\Leftrightarrow wx_2 \in \{w_2 x_2\} \\
&\Leftrightarrow wx_2 \in (\lambda \mathbb{W} (w_2 x_2)).
\end{aligned}$$

$$\begin{aligned}
&\text{case 2: } w_1 = \bar{w}_1 a \text{ with } \bar{w}_1 \in \mathcal{R}_*, a \in \mathcal{R}_*: \\
wx_2 \in ((\bar{w}_1 a) \mathbb{W} w_2) \{x_2\} &\Leftrightarrow wx_2 \in ((\bar{w}_1 a) \mathbb{W} w_2) \{x_2\} \cup \\
&\quad (\bar{w}_1 \mathbb{W} w_2) \{a+x_2\} \cup \\
&\quad (\bar{w}_1 \mathbb{W} (w_2 x_2)) \{a\} \\
&\Leftrightarrow wx_2 \in ((\bar{w}_1 a) \mathbb{W} (w_2 x_2)).
\end{aligned}$$

(\*) ' $\Leftarrow$ ' holds since  $w \in w_1 \mathbb{W} w_2$

□

Lemma 58 states when having two weakly composed rule sequences one can, analogously but reverse to the definition of the weak parallel composition, also split off rules from the right side of the sequences. I.e. the weak parallel composition of rule sequences treats left and right addition equally.

**Lemma 58.** Let  $I \in \mathcal{R}_*$  be a set of parallel rules. Let  $w_1, w_2 \in \mathcal{R}_*$ , and  $r_1, r_2 \in \mathcal{R}_*$ . Then the following holds:

$$(w_1 r_1) \mathbb{W} (w_2 r_2) = (w_1 \mathbb{W} w_2) \{r_1+r_2\} \cup (w_1 \mathbb{W} (w_2 r_2)) \{r_1\} \cup ((w_1 r_1) \mathbb{W} w_2) \{r_2\}.$$

*Proof.* Induction over  $|w_1| + |w_2|$

*Basis:*  $|w_1| + |w_2| = 0$ :

$|w_1| + |w_2| = 0$  implies  $w_1 = \lambda, w_2 = \lambda$ .

$$\begin{aligned}
(\lambda r_1) \mathbb{W} (\lambda r_2) &= r_1 \mathbb{W} r_2 \\
&= \{r_1+r_2\} \cup \{r_1; r_2\} \cup \{r_2; r_1\} && [\text{def. 39(iii)}]
\end{aligned}$$

$$\begin{aligned}
&= \{\lambda\}\{r_1+r_2\} \cup \{r_1\}\{r_2\} \cup \{r_2\}\{r_1\} && [\text{def. 41, prop. 43}] \\
&= (\lambda \mathbb{W} \lambda)\{r_1+r_2\} \cup (r_1 \mathbb{W} \lambda)\{r_2\} \cup (\lambda \mathbb{W} r_2)\{r_1\} && [\text{def. 39(i)}] \\
&= (\lambda \mathbb{W} \lambda)\{r_1+r_2\} \cup (\lambda \mathbb{W} (\lambda r_2))\{r_1\} \cup ((\lambda r_1) \mathbb{W} \lambda)\{r_2\}.
\end{aligned}$$

*Hypothesis:* For all  $w_1, w_2$  with  $|w_1| + |w_2| \leq n, n \in \mathbb{N}$  holds:

$$(w_1 r_1) \mathbb{W} (w_2 r_2) = (w_1 \mathbb{W} w_2)\{r_1+r_2\} \cup (w_1 \mathbb{W} (w_2 r_2))\{r_1\} \cup ((w_1 r_1) \mathbb{W} w_2)\{r_2\}$$

*Step:* Step from  $n$  to  $n+1$  implies  $|xw_1| + |w_2| = n+1$  or  $|w_1| + |xw_2| = n+1$  with  $x \in I$ .

W.l.o.g. we choose  $|xw_1| + |w_2| = n+1$ .

Case 1:  $w_2 = \lambda$ :

$$\begin{aligned}
(xw_1 r_1) \mathbb{W} (\lambda r_2) &= (xw_1 r_1) \mathbb{W} r_2 \\
&= \{x+r_2\}((w_1 r_1) \mathbb{W} \lambda) \cup \\
&\quad \{x\}((w_1 r_1) \mathbb{W} r_2) \cup \\
&\quad \{r_2\}((xw_1 r_1) \mathbb{W} \lambda) && [\text{def. 39(iii)}] \\
&= \{x_1+r_2\}((w_1 r_1) \mathbb{W} \lambda) \cup && [\text{i.h.}] \\
&\quad \{x\}((w_1 \mathbb{W} \lambda)\{r_1+r_2\} \cup (w_1 \mathbb{W} r_2)\{r_1\} \cup (w_1 r_1 \mathbb{W} \lambda)\{r_2\}) \cup \\
&\quad \{r_2\}((xw_1 r_1) \mathbb{W} \lambda) \\
&= \{x_1+r_2\}\{w_1 r_1\} \cup && [\text{def. 39(i)}] \\
&\quad \{x\}(\{w_1\}\{r_1+r_2\} \cup (w_1 \mathbb{W} r_2)\{r_1\} \cup \{w_1 r_1\}\{r_2\}) \cup \\
&\quad \{r_2\}\{xw_1 r_1\} \\
&= \{x_1+r_2\}\{w_1\}\{r_1\} \cup \\
&\quad \{x\}\{w_1\}\{r_1+r_2\} \cup \{x\}(w_1 \mathbb{W} r_2)\{r_1\} \cup \{x\}\{w_1 r_1\}\{r_2\} \cup \\
&\quad \{r_2\}\{xw_1\}\{r_1\} \\
&= \{x_1+r_2\}\{w_1 \mathbb{W} \lambda\}\{r_1\} \cup && [\text{def. 39(iii)}] \\
&\quad \{x\}\{w_1\}\{r_1+r_2\} \cup \{x\}(w_1 \mathbb{W} r_2)\{r_1\} \cup \{x\}\{w_1 r_1\}\{r_2\} \cup \\
&\quad \{r_2\}\{xw_1 \mathbb{W} \lambda\}\{r_1\} \\
&= \{x\}\{w_1\}\{r_1+r_2\} \cup \\
&\quad \{x_1+r_2\}\{w_1 \mathbb{W} \lambda\}\{r_1\} \cup \{x\}(w_1 \mathbb{W} r_2)\{r_1\} \cup \{r_2\}\{xw_1 \mathbb{W} \lambda\}\{r_1\} \cup \\
&\quad \{x\}\{w_1 r_1\}\{r_2\} \\
&= \{xw_1\}\{r_1+r_2\} \cup \\
&\quad (\{x_1+r_2\}\{w_1 \mathbb{W} \lambda\} \cup \{x\}(w_1 \mathbb{W} r_2) \cup \{r_2\}\{xw_1 \mathbb{W} \lambda\})\{r_1\} \cup \\
&\quad \{xw_1 r_1\}\{r_2\} \\
&= (xw_1 \mathbb{W} \lambda)\{r_1+r_2\} \cup && [\text{def. 39 (i),(iii)}] \\
&\quad (xw_1 \mathbb{W} \lambda r_2)\{r_1\} \cup \\
&\quad (xw_1 r_1 \mathbb{W} \lambda)\{r_2\}.
\end{aligned}$$

Case 2:  $w_2 = b\bar{w}_2$ :

$$\begin{aligned}
(xw_1 r_1) \mathbb{W} (b\bar{w}_2 r_2) &= \{x+b\}((w_1 r_1) \mathbb{W} (\bar{w}_2 r_2)) \cup && [\text{def. 39}] \\
&\quad \{x\}((w_1 r_1) \mathbb{W} (b\bar{w}_2 r_2)) \cup \\
&\quad \{b\}((xw_1 r_1) \mathbb{W} (\bar{w}_2 r_2))
\end{aligned}$$

$$\begin{aligned}
&= \{x+b\}((w_1 \mathbb{W} \bar{w}_2)\{r_1+r_2\} \cup (w_1 \mathbb{W}(\bar{w}_2 r_2))\{r_1\} \cup ((w_1 r_1) \mathbb{W} \bar{w}_2)\{r_2\}) \cup \\
&\quad \{x\}((w_1 \mathbb{W}(b\bar{w}_2))\{r_1+r_2\} \cup (w_1 \mathbb{W}(b\bar{w}_2 r_2))\{r_1\} \cup ((w_1 r_1) \mathbb{W}(b\bar{w}_2))\{r_2\}) \cup \\
&\quad \{b\}((xw_1) \mathbb{W} \bar{w}_2)\{r_1+r_2\} \cup ((xw_1) \mathbb{W}(\bar{w}_2 r_2))\{r_1\} \cup ((xw_1 r_1) \mathbb{W} \bar{w}_2)\{r_2\}) \text{ [i.h.]} \\
&= \{x+b\}(w_1 \mathbb{W} \bar{w}_2)\{r_1+r_2\} \cup \{x+b\}(w_1 \mathbb{W}(\bar{w}_2 r_2))\{r_1\} \cup \{x+b\}((w_1 r_1) \mathbb{W} \bar{w}_2)\{r_2\} \cup \\
&\quad \{x\}(w_1 \mathbb{W}(b\bar{w}_2))\{r_1+r_2\} \cup \{x\}(w_1 \mathbb{W}(b\bar{w}_2 r_2))\{r_1\} \cup \{x\}((w_1 r_1) \mathbb{W}(b\bar{w}_2))\{r_2\} \cup \\
&\quad \{b\}((xw_1) \mathbb{W} \bar{w}_2)\{r_1+r_2\} \cup \{b\}((xw_1) \mathbb{W}(\bar{w}_2 r_2))\{r_1\} \cup \{b\}((xw_1 r_1) \mathbb{W} \bar{w}_2)\{r_2\} \\
&= \{x+b\}(w_1 \mathbb{W} \bar{w}_2)\{r_1+r_2\} \cup \{x\}(w_1 \mathbb{W}(b\bar{w}_2))\{r_1+r_2\} \cup \{b\}((xw_1) \mathbb{W} \bar{w}_2)\{r_1+r_2\} \cup \\
&\quad \{x+b\}(w_1 \mathbb{W}(\bar{w}_2 r_2))\{r_1\} \cup \{x\}(w_1 \mathbb{W}(b\bar{w}_2 r_2))\{r_1\} \cup \{b\}((xw_1) \mathbb{W}(\bar{w}_2 r_2))\{r_1\} \cup \\
&\quad \{x+b\}((w_1 r_1) \mathbb{W} \bar{w}_2)\{r_2\} \cup \{x\}((w_1 r_1) \mathbb{W}(b\bar{w}_2))\{r_2\} \cup \{b\}((xw_1 r_1) \mathbb{W} \bar{w}_2)\{r_2\} \\
&= (\{x+b\}(w_1 \mathbb{W} \bar{w}_2) \cup \{x\}(w_1 \mathbb{W}(b\bar{w}_2)) \cup \{b\}((xw_1) \mathbb{W} \bar{w}_2))\{r_1+r_2\} \cup \\
&\quad (\{x+b\}(w_1 \mathbb{W}(\bar{w}_2 r_2)) \cup \{x\}(w_1 \mathbb{W}(b\bar{w}_2 r_2)) \cup \{b\}((xw_1) \mathbb{W}(\bar{w}_2 r_2)))\{r_1\} \cup \\
&\quad (\{x+b\}((w_1 r_1) \mathbb{W} \bar{w}_2) \cup \{x\}((w_1 r_1) \mathbb{W}(b\bar{w}_2)) \cup \{b\}((xw_1 r_1) \mathbb{W} \bar{w}_2))\{r_2\} \cup \\
&= ((xw_1) \mathbb{W}(b\bar{w}_2))\{r_1+r_2\} \cup \text{ [def. 39]} \\
&\quad ((xw_1) \mathbb{W}(b\bar{w}_2 r_2))\{r_1\} \cup \\
&\quad ((xw_1 r_1) \mathbb{W}(b\bar{w}_2))\{r_2\}.
\end{aligned}$$

□

### A little excursion regarding weak parallel composition of words

Lemma 58 suggests that considering two words to compose weak parallel it makes no difference if they are composed from the left side (forward) or from the right side (backward). In order to prove this assumption we define backward weak parallel composition (denoted by  $\overleftarrow{\mathbb{W}}$ ) and then show that forward and backward parallel composition have the same result.

#### Definition 59. Backwards weak parallel composition

Let  $I$  be an alphabet,  $w, w_1, w_2 \in I^*$ , and  $r_1, r_2 \in I$ . The backwards weak parallel composition  $\overleftarrow{\mathbb{W}}$  of  $w_1$  and  $w_2$  is defined by

$$\begin{aligned}
w \overleftarrow{\mathbb{W}} \lambda &= \{w\} = \lambda \overleftarrow{\mathbb{W}} w \\
w_1 r_1 \overleftarrow{\mathbb{W}} w_2 r_2 &= (w_1 \overleftarrow{\mathbb{W}} w_2)\{r_1+r_2\} \cup (w_1 \overleftarrow{\mathbb{W}}(w_2 r_2))\{r_1\} \cup ((w_1 r_1) \overleftarrow{\mathbb{W}} w_2)\{r_2\}
\end{aligned}$$

**Proposition 60.** Let  $I$  be an alphabet and let  $w_1, w_2 \in I^*$ . Then it holds  $w_1 \mathbb{W} w_2 = w_1 \overleftarrow{\mathbb{W}} w_2$ .

*Proof.* We prove the proposition by induction over  $|w_1| + |w_2|$ .

*Basis:*  $|w_1| + |w_2| = 0$ :

$|w_1| + |w_2| = 0$  implies  $w_1 = \lambda$  and  $w_2 = \lambda$ .

$$\begin{aligned}
\lambda \mathbb{W} \lambda &= \lambda && \text{[def. 39(i)]} \\
&= \lambda \overleftarrow{\mathbb{W}} \lambda. && \text{[def. 59]}
\end{aligned}$$

*Hypothesis:*  $w_1 \mathbb{W} w_2 = w_1 \overleftarrow{\mathbb{W}} w_2$  holds for all  $w_1, w_2$  with  $|w_1| + |w_2| \leq n$ .

*Step:* Proof for  $w'_1 \mathbb{W} w'_2$  with  $|w'_1| + |w'_2| = n + 1$ .

$|w'_1| + |w'_2| = n + 1$  implies  $w'_1 = w_1x, w'_2 = w_2$  or  $w'_1 = w_1, w'_2 = w_2y$ .

W.l.o.g. we choose  $w'_1 = w_1x, w'_2 = w_2$ .

Case 1:  $w_2 = \lambda$ :

$$\begin{aligned} w_1x \mathbb{W}\lambda &= w_1x && \text{[def. 39(i)]} \\ &= w_1x \overleftarrow{\mathbb{W}}\lambda. && \text{[def. 59]} \end{aligned}$$

Case 2:  $w_2 = \bar{w}_2y$ :

$$\begin{aligned} w_1x \mathbb{W}\bar{w}_2y &= (w_1 \mathbb{W}\bar{w}_2)\{x+y\} \cup (w_1 \mathbb{W}(\bar{w}_2y))\{x\} \cup ((w_1x) \mathbb{W}\bar{w}_2)\{y\} && \text{[lemma 58]} \\ &= (w_1 \overleftarrow{\mathbb{W}}\bar{w}_2)\{x+y\} \cup (w_1 \overleftarrow{\mathbb{W}}(\bar{w}_2y))\{x\} \cup ((w_1x) \overleftarrow{\mathbb{W}}\bar{w}_2)\{y\} && \text{[i.h.]} \\ &= w_1x \overleftarrow{\mathbb{W}}\bar{w}_2y && \text{[def. 59]} \quad \square \end{aligned}$$

### Synchronous case

As for the weak parallel case we now prove the induction step for the synchronous case. The proof is analogously constructed and employs analogous lemmas also given subsequently to the needing proof.

**Proposition 61.** Let  $C_1$  and  $C_2$  be two parallel expressions, and let by induction hypothesis  $L(A(C_1)) = L(C_1)$  and  $L(A(C_2)) = L(C_2)$ . Then it holds  $L(A(C_1\$C_2)) = L(C_1\$C_2)$ .

*Proof.*

Let  $A(C_i) = (S_i, I_i, d_i, s_{0_i}, F_i), i \in \{1, 2\}$  be the two finite state automata for  $C_1$  respectively  $C_2$ . Moreover, let

$$A(C_1\$C_2) = A(C_1)\$A(C_2) = (S, I, d_{\#}, (s_{0_1}, s_{0_2}), F).$$

$$\begin{aligned} L(A(C_1\$C_2)) &= L(A(C_1)\$A(C_2)) && \text{[def. 52]} \\ &= \{w \mid ((s_{0_1}, s_{0_2}), w, (f_1, f_2)) \in d_{\#}^*, (f_1, f_2) \in F && \text{[def. } L(A)] \\ &\quad \text{or} \\ &\quad ((s_{0_1}, s_{0_2}), w, f_1) \in d_{\#}^*, f_1 \in F_1 \\ &\quad \text{or} \\ &\quad ((s_{0_1}, s_{0_2}), w, f_2) \in d_{\#}^*, f_2 \in F_2\} \\ &= \{w_1\$w_2 \mid (s_{0_1}, w_1, f_1) \in d_1^*, (s_{0_2}, w_2, f_2) \in d_2^*, f_1 \in F_1, f_2 \in F_2 && \text{[lemma 62]} \\ &\quad \text{or} \\ &\quad (s_{0_1}, w_1, f_1) \in d_1^*, \exists (s_{0_2}, w_2, s_2) \in d_2^*, s_2 \in F_2, f_1 \in F_1 \\ &\quad \text{or} \\ &\quad \exists (s_{0_1}, w_1, s_1) \in d_1^*, s_1 \in F_1, (s_{0_2}, w_2, f_2) \in d_2^*, f_2 \in F_2\} \\ &= \{w_1\$w_2 \mid w_1 \in L(A(C_1)), w_2 \in L(A(C_2))\} && \text{[} f_1, s_1 \in F_1, f_2, s_2 \in F_2 \\ &= L(A(C_1))\$L(A(C_2)) && \text{[def. 40]} \\ &= L(C_1)\$L(C_2) && \text{[i.h.]} \\ &= L(C_1\$C_2). && \text{[def. 41]} \quad \square \end{aligned}$$

Analogously to Lemma 56 the following Lemma 62 states that whenever a synchronous composition automaton  $A_1\$A_2$  processes a rule sequence  $w$  there are rule sequences  $w_1$  and  $w_2$  processed by  $A_1$  respectively  $A_2$  and  $w$  is the synchronous composition of  $w_1$  and  $w_2$ . Lemma 62 consists of three parts due to the construction of the synchronous composition automata. Part a) addresses the case where both input automata yet take part in the creation of  $w$ , i.e.  $w$  is comprised of  $w_1$  and  $w_2$  in equal shares. Part b) covers the case where at some point of the creation of  $w$   $A_2$  has reached a final state and  $A_1$  operates further alone, i.e. the first part of  $w$  is composed of  $w'_1$  and  $w_2$  with  $w_1 = w'_1w''_1$  and the last part consists of  $w''_1$ . Part c) covers the analogous situation for  $A_1$  has reached a final state. Since the proof for Lemma 62 needs information about the ratio of the length of  $w_1$  and  $w_2$  ( $|w_1| = |w_2|$  respectively  $|w_1| < |w_2|$ , and  $|w_1| > |w_2|$ ) we add this information to the lemma.

**Lemma 62.** Let  $A_i = (S_i, I_i, d_i, s_{i0}, F_i), i \in \{1, 2\}$  with  $I_i \in \mathcal{R}_*$  be two finite state automata and  $A = (S, I, d_{\#}, (s_{01}, s_{02}), F)$  be the synchronous composition  $A_1\$A_2$ . Then for all  $w \in I_*^*$  holds:

- a)  $(s'_1, s'_2) \in d_{\#}^*((s_1, s_2), w) \Leftrightarrow w = w_1\$w_2,$   
 $s'_1 \in d_1^*(s_1, w_1), s'_2 \in d_2^*(s_2, w_2),$   
 $|w_1| = |w_2|,$
- b)  $s'_1 \in d_{\#}^*((s_1, s_2), w), s'_1 \in S_1 \Leftrightarrow w = w_1\$w_2,$   
 $s'_1 \in d_1^*(s_1, w_1), \exists s'_2 \in d_2^*(s_2, w_2), s'_2 \in F_2,$   
 $|w_1| > |w_2|,$
- c)  $s'_2 \in d_{\#}^*((s_1, s_2), w), s'_2 \in S_2 \Leftrightarrow w = w_1\$w_2,$   
 $\exists s'_1 \in d_1^*(s_1, w_1), s'_1 \in F_1, s'_2 \in d_2^*(s_2, w_2),$   
 $|w_1| < |w_2|.$

*Proof.*

a) Induction over the structure of  $w$ .

*Basis:*  $w = \lambda$ :

$$(s'_1, s'_2) \in d_{\#}^*((s_1, s_2), \lambda) \Leftrightarrow d_1^*(s_1, \lambda) = \{s'_1\}, d_2^*(s_2, \lambda) = \{s'_2\},$$

$$\lambda\$ \lambda = \lambda, |\lambda| = |\lambda|.$$

*Hypothesis:* For a  $w \in I_*^*$  holds:

$$(s'_1, s'_2) \in d_{\#}^*((s_1, s_2), w) \Leftrightarrow w = w_1\$w_2,$$

$$s'_1 \in d_1^*(s_1, w_1), s'_2 \in d_2^*(s_2, w_2),$$

$$|w_1| = |w_2|.$$

*Step:* Proof for  $wx$  with  $x \in I_*$ .

$$\begin{aligned}
(s'_1, s'_2) \in d_{\sharp}^*((s_1, s_2), wx) &\Leftrightarrow (s'_1, s'_2) \in d_{\sharp}((\bar{s}_1, \bar{s}_2), x) && [\text{def. } d^*] \\
&\quad \text{for some } (\bar{s}_1, \bar{s}_2) \in d_{\sharp}^*((s_1, s_2), w) \\
&\Leftrightarrow (s'_1, s'_2) \in d_{\sharp}((\bar{s}_1, \bar{s}_2), x) && [\text{i.h.}] \\
&\quad \text{for some } \bar{s}_1 \in d_1^*(s_1, w_1), \bar{s}_2 \in d_2^*(s_2, w_2), \\
&\quad |w_1| = |w_2|, w = w_1\$w_2
\end{aligned}$$

Knowing that  $(s'_1, s'_2) \in d_{\sharp}((\bar{s}_1, \bar{s}_2), x)$  is a state transition in  $A_1\$A_2$  we can, employing  $d_{\sharp}$ , infer that there are state transitions  $s'_1 \in d_1(\bar{s}_1, x_1)$ ,  $s'_2 \in d_2(\bar{s}_2, x_2)$  in  $A_1$  resp.  $A_2$  with  $x = x_1 + x_2$ .

$$\begin{aligned}
&\Leftrightarrow s'_1 \in d_1(\bar{s}_1, x_1), s'_2 \in d_2(\bar{s}_2, x_2), x = x_1 + x_2, \\
&\quad \text{for some } \bar{s}_1 \in d_1^*(s_1, w_1), \bar{s}_2 \in d_2^*(s_2, w_2), \\
&\quad |w_1| = |w_2|, w = w_1\$w_2 && [\text{def. } d_{\sharp}]
\end{aligned}$$

Now we are able to combine the state transitions  $d_1$  processing  $x_1$  with  $d_1^*$  processing  $w_1$  resp.  $d_2$  processing  $x_2$  with  $d_2^*$  processing  $w_2$ .

$$\begin{aligned}
&\Leftrightarrow s'_1 \in d_1^*(s_1, wx_1), s'_2 \in d_2^*(s_2, wx_2), x = x_1 + x_2, \\
&\quad |w_1| = |w_2|, w = w_1\$w_2 && [\text{def. } d^*]
\end{aligned}$$

Since we know that  $|w_1| = |w_2|$  we also know that  $|w_1x_1| = |w_2x_2|$ . With the help of Lemma 63a we infer that  $wx$  processed by  $A_1\$A_2$  is the synchronous composition of  $w_1x_1$  processed by  $A_1$  and  $w_2x_2$  processed by  $A_2$

$$\Leftrightarrow s'_1 \in d_1^*(s_1, wx_1), s'_2 \in d_2^*(s_2, wx_2).$$

Moreover,  $((|w_1| = |w_2|) \Leftrightarrow (|w_1x_1| = |w_2x_2|))$ , and

$$\begin{aligned}
wx &= (w_1\$w_2)(x_1 + x_2) \\
&= (w_1x_1)\$(w_2x_2). && [\text{lemma 63(a)}]
\end{aligned}$$

b) Induction over  $|w_1| - |w_2|$ .

Since b) covers the case where at some point of the creation of  $w$   $A_2$  has reached a final state and  $A_1$  operates further alone we run the induction over  $|w_1| - |w_2|$  where  $|w_1| - |w_2| = 1$  is the induction basis (the first time where  $A_1$  operates alone). Since  $w_1$  and  $w_2$  only occur on the right side of the equation to be proved we begin the proof from there.

*Basis:*  $|w_1| - |w_2| = 1$ :

$|w_1| - |w_2| = 1$  implies  $w_1 = w'_1y$  with  $|w'_1| = |w_2|$ .

$$\begin{aligned}
&s'_1 \in d_1^*(s_1, w'_1y), s'_2 \in d_2^*(s_2, w_2), s'_2 \in F_2, w = (w'_1y)\$w_2, |w'_1| = |w_2| \\
&\Leftrightarrow s'_1 \in d_1(\bar{s}_1, y) \text{ for some } \bar{s}_1 \in d_1^*(s_1, w'_1), s'_2 \in d_2^*(s_2, w_2), s'_2 \in F_2, \\
&\quad w = (w'_1y)\$w_2, |w'_1| = |w_2| && [d^*] \\
&\Leftrightarrow s'_1 \in d_1(\bar{s}_1, y) \text{ for some } (\bar{s}_1, s'_2) \in d_{\sharp}^*((s_1, s_2), w'_1\$w_2), s'_2 \in F_2, \\
&\quad w = (w'_1y)\$w_2 \text{ and } s'_1 \in S_1 \text{ since } s_1 \in d_1(\bar{s}_1, y) && [\text{lemma 62 (a)}] \\
&\Leftrightarrow s'_1 \in d_{\sharp}((\bar{s}_1, s_2), y) \text{ for some } (\bar{s}_1, s'_2) \in d_{\sharp}^*((s_1, s_2), w'_1\$w_2), s'_1 \in S_1 && [d_{\sharp}] \\
&\Leftrightarrow s'_1 \in d_{\sharp}^*((s_1, s_2), (w'_1\$w_2)y), s'_1 \in S_1 && [d^*] \\
&\Leftrightarrow s'_1 \in d_{\sharp}^*((s_1, s_2), (w'_1y)\$w_2), s'_1 \in S_1. && [\text{lemma 63(b)}]
\end{aligned}$$

*Hypothesis:* For all  $w_1, w_2$  with  $|w_1| - |w_2| = n$  for an  $n \in \mathbb{N}$  holds:

$$\begin{aligned}
s'_1 \in d^*((s_1, s_2), w), s'_1 \in S_1 &\Leftrightarrow w = w_1\$w_2, \\
& s'_1 \in d_1^*(s_1, w_1), \exists s'_2 \in d_2^*(s_2, w_2), s'_2 \in F_2, \\
& |w_1| > |w_2|.
\end{aligned}$$

*Step:* Step from  $|w_1| - |w_2| = n$  to  $|w'_1| - |w'_2| = n + 1$ .  
 $|w_1| - |w_2| = n, |w'_1| - |w'_2| = n + 1$  implies  $w'_1 = w_1x, w'_2 = w_2$  or  $w'_1 = w_1, w'_2 = w_2x$  for  $x \in \mathcal{R}_*$ . W.l.o.g. we choose  $w'_1 = w_1x, w'_2 = w_2$ .  
 $s'_1 \in d_1^*(s_1, w_1x), s'_2 \in d_2^*(s_2, w_2), s'_2 \in F_2, |w_1x| > |w_2|, w = (w_1x)\$w_2$   
 $\Leftrightarrow s'_1 \in d_1(\bar{s}_1, x)$  for some  $\bar{s}_1 \in d_1^*(s_1, w_1), s'_2 \in d_2^*(s_2, w_2), s'_2 \in F_2,$   
 $|w_1x| > |w_2|, w = (w_1x)\$w_2$  [d\*]  
 $\Leftrightarrow s'_1 \in d_1(\bar{s}_1, x)$  for some  $\bar{s}_1 \in d_{\#}^*((s_1, s_2), w_1\$w_2), s'_2 \in F_2, w = (w_1x)\$w_2.$   
Moreover,  $s'_1 \in S_1$  since  $s'_1 \in d_1(\bar{s}_1, x)$  [i.h.]  
 $\Leftrightarrow s'_1 \in d_{\#}(\bar{s}_1, x)$  for some  $\bar{s}_1 \in d_{\#}^*((s_1, s_2), w_1\$w_2), s'_2 \in F_2, w = (w_1x)\$w_2, s'_1 \in S_1$  [d\_{\#}]  
 $\Leftrightarrow s'_1 \in d_{\#}^*((s_1, s_2), (w_1\$w_2)x), s'_1 \in S_1$  [d\*]  
 $\Leftrightarrow s'_1 \in d_{\#}^*((s_1, s_2), (w_1x)\$w_2), s'_1 \in S_1.$  [lemma 63(b)]

c) The proof for c) is analogous to b). □

The following Lemma 63 is needed in the last steps of the above proofs for Lemma 62 in order to show that  $x_1+x_2$  respectively  $x$  when sequentially composed to  $w_1\$w_2$  could be involved in the synchronous composition in the desired position. According to the three parts of Lemma 62 Lemma 63 also is comprised of three parts. Each part representing one possible situation when composing two words synchronously, employing the information about the ratio of the length of the words to each other.

**Lemma 63.** Let  $\mathcal{R}_*$  be a class of parallel rules. Moreover, let  $w_1, w_2 \in \mathcal{R}_*$  and  $x_1, x_2, \bar{x} \in \mathcal{R}_*$ . Then

- a)  $(w_1\$w_2)(x_1+x_2) = (w_1x_1)\$(w_2x_2)$  if  $|w_1| = |w_2|$ ,
- b)  $(w_1\$w_2)\bar{x} = (w_1\bar{x})\$w_2$  if  $|w_1| \geq |w_2|$ ,
- c)  $(w_1\$w_2)\bar{x} = w_1\$(w_2\bar{x})$  if  $|w_1| \leq |w_2|$ .

*Proof.*

a) Induction over the structure of  $w_1$ .

*Basis:*  $w_1 = \lambda$ :

Since  $|w_1| = |w_2|$  and  $w_1 = \lambda$  we obtain  $w_2 = \lambda$ .

$$\begin{aligned}
(\lambda\$ \lambda)(x_1 + x_2) &= \lambda(x_1 + x_2) && \text{[def. 39(b(i))]} \\
&= (x_1 + x_2)\lambda \\
&= (x_1 + x_2)(\lambda\$ \lambda) && \text{[def. 39(b(i))]} \\
&= (x_1\lambda)\$(x_2\lambda) && \text{[def. 39(b(ii))]} \\
&= x_1\$x_2
\end{aligned}$$

$$= (\lambda x_1)\$(\lambda x_2).$$

*Hypothesis:* For a  $w_1 \in \mathcal{R}_*$  holds:

$$(w_1\$w_2)(x_1 + x_2) = (w_1x_1)\$(w_2x_2) \text{ if } |w_1| = |w_2|.$$

*Step:* Proof for  $xw_1$  with  $x \in \mathcal{R}_*$ :

$|xw_1| = |w_2|$  implies  $w_2 = yw'_2$ . Then we obtain

$$\begin{aligned} ((xw_1)\$(yw'_2))(x_1 + x_2) &= (x + y)(w_1\$w'_2)(x_1 + x_2) && [\text{def. 39(b(ii))}] \\ &= (x + y)((w_1x_1)\$(w'_2x_2)) && [\text{i.h.}] \\ &= (xw_1x_1)\$(yw'_2x_2) && [\text{def. 39(b(ii))}] \\ &= (xw_1x_1)\$(w_2x_2). \end{aligned}$$

b) Induction over the structure of  $w_1$ .

*Basis:*  $w_1 = \lambda$ :

Since  $|w_1| \geq |w_2|$ , and  $w_1 = \lambda$  we obtain  $w_2 = \lambda$ .

$$(\lambda\$ \lambda)x = \lambda x = (\lambda x)\$ \lambda.$$

*Hypothesis:* For a  $w \in \mathcal{R}_*$  holds:

$$(w_1\$w_2)\bar{x} = (w_1\bar{x})\$w_2 \text{ if } |w_1| \geq |w_2|.$$

*Step:* Proof for  $xw_1$  with  $x \in \mathcal{R}_*$ .

case 1:  $w_2 = \lambda$ :

$$\begin{aligned} ((xw_1)\$ \lambda)\bar{x} &= (xw_1)\bar{x} && [\text{def. 39(b(i))}] \\ &= xw_1\bar{x} \\ &= (xw_1\bar{x})\$ \lambda. && [\text{def. 39(b(i))}] \end{aligned}$$

case 2:  $w_2 = yw'_2$ :

$$\begin{aligned} ((xw_1)\$(yw'_2))\bar{x} &= ((x + y)(w_1\$w'_2))\bar{x} && [\text{def. 39(b(ii))}] \\ &= (x + y)((w_1\$w'_2)\bar{x}) && [\text{ass. concat.}] \end{aligned}$$

Since  $|w_1| \geq |w_2|$  we also know that  $w_1 \geq w'_2$  because  $w_2 = w'_2y$ . So we are able to apply the induction hypothesis

$$\begin{aligned} &= (x + y)((w_1\bar{x})\$w'_2) && [\text{i.h.}] \\ &= (xw_1\bar{x})\$(yw'_2) && [\text{def. 39(b(ii))}] \\ &= (xw_1\bar{x})\$(w_2). \end{aligned}$$

c) The proof for c) is analogous to b). □



# Chapter 5

## As-long-as-possible Control Condition

This chapter introduces an as-long-as-possible control condition which is able to express the iteration of an entire process until the process could not take place anymore entirely.

In the Preliminaries we have encountered an as-long-as-possible control condition over a set of rules, demanding the application of rules from the set until no more application is possible. As a reminder, the semantics of that as-long-as-possible control condition is given by:  $SEM(as-long-as-possible) = \{(G, H) \mid G \xrightarrow{*} H \in der(P), \nexists H \xrightarrow[r]{} H', \forall r \in P\}$ .

In the following we enhance this concept to entire expressions rather than a set of rules. The resulting control condition is called *as-long-as-possible-expression*.

### 5.1 As-long-as-possible Expressions

As-long-as-possible expressions (*alap*-expressions for short) enhance regular expressions by an unary as-long-as-possible operator, denoted by the symbol  $!$ . The operator  $!$  applied to an expressions  $e$  means that  $e$  has to be iteratively executed as long as there are rules, prescribed by  $e$ , that can be applied to the current graph. The execution stops if no more iteration of  $e$  is possible.

## 5.2 Syntax and Semantics

The syntax of *alap*-expressions over a given rule set is, apart from the operator  $!$ , defined like for regular expressions.

### Definition 64. *Alap*-expressions over rules

Let  $P \subseteq \mathcal{R}$  be a set of rules. *Alap*-expressions over  $P$  are recursively defined by

- (i)  $\emptyset$ ,  $\lambda$ , and  $r \in P$  are *alap*-expressions,
- (ii) if  $C, C_1$  and  $C_2$  are *alap*-expressions,  $C_1; C_2$ ,  $C_1|C_2$ ,  $C^*$ , and  $C!$  are *alap*-expressions.

The binding strength for the operators  $^*$ ,  $;$ , and  $|$  is the same as for regular expressions, and  $!$  has the same binding strength as  $^*$ .

The semantics of an *alap*-expression is defined as a set of derivations. Other than for regular expressions, the semantics of *alap*-expressions depends on the actual input graph. Therefore we provide two semantic definitions. The first explicitly takes into account a given input graph. In the second definition all graphs from  $\mathcal{G}$  are considered as input graphs. The operators used within the definitions are defined afterwards.

### Definition 65. Semantics of *alap*-expressions for a given input graph

Let  $P \subseteq \mathcal{R}$ ,  $r \in P$  and  $C, C_1$ , and  $C_2$  be *alap*-expressions. Moreover let  $G \in \mathcal{G}$  be a graph. The semantics of an *alap*-expression provided the input graph  $G$  is defined by

- $SEM(\emptyset, G) = \emptyset$ ,
- $SEM(\lambda, G) = \{G \xRightarrow{\emptyset} G\}$ ,
- $SEM(r, G) = \{G \xRightarrow{r} H \mid H \in \mathcal{G}\}$ ,
- $SEM(C_1; C_2, G) = SEM(C_1, G) \circ \bigcup_{G \xRightarrow{*} G' \in SEM(C_1, G)} SEM(C_2, G')$ ,
- $SEM(C_1|C_2, G) = SEM(C_1, G) \cup SEM(C_2, G)$ ,
- $SEM(C^*, G) = \bigcup_{i=0}^{\infty} SEM(C, G)^i$ , and
- $SEM(C!, G) = \{G \xRightarrow{*} G' \in SEM(C^*, G) \mid SEM(C, G') = \emptyset\} \cup SEM(C, G)^\infty$ .

Considering no specific input graph, but all possible graphs, the semantics of an *alap*-expression is defined as follows.

**Definition 66. Semantics of *alap*-expressions**

Let  $P \subseteq \mathcal{R}$ ,  $r \in P$  and  $C, C_1$ , and  $C_2$  be *alap*-expressions.

- $SEM(\emptyset) = \emptyset$ ,
- $SEM(\lambda) = \{G \xrightarrow{0} G \mid G \in \mathcal{G}\}$ ,
- $SEM(r) = \{G \xrightarrow[r]{} H \mid G, H \in \mathcal{G}\}$ ,
- $SEM(C_1; C_2) = SEM(C_1) \circ SEM(C_2)$ ,
- $SEM(C_1|C_2) = SEM(C_1) \cup SEM(C_2)$ ,
- $SEM(C^*) = \bigcup_{i=0}^{\infty} SEM(C)^i$ , and
- $SEM(C!) = \{G \xrightarrow{*} G' \in SEM(C^*) \mid G' \xrightarrow{*} G'' \notin SEM(C), G, G', G'' \in \mathcal{G}\} \cup SEM(C)^\infty$ .

In the following the iterative operators used in the semantic definitions are defined.

**Definition 67. Finite and infinite iteration**

Let  $C$  be a *alap*-expression and  $G \in \mathcal{G}$ .

The  $n$ -times iteration of the semantics of  $C$ , for an  $n \in \mathbb{N}$ , according to an input graph  $G$  is recursively defined by:

- (i)  $SEM(C, G)^0 = \{G \xrightarrow{0} G\}$ ,
- (ii)  $SEM(C, G)^{n+1} = SEM(C, G) \circ \{SEM(C, G')^n \mid G \xrightarrow{*} G' \in SEM(C, G)\}$ .

The infinite iteration is defined by:

$$SEM(C, G_0)^\infty = \{G_0 \xrightarrow{*} G_1 \xrightarrow{*} G_2 \xrightarrow{*} \dots \mid G_i \xrightarrow{*} G_{i+1} \in SEM(C, G_i), G_i \in \mathcal{G}, i \in \mathbb{N}\}.$$

Considering all graphs as input graphs the  $n$ -times iteration is defined by:

- (i)  $SEM(C)^0 = \{G \xrightarrow{0} G \mid G \in \mathcal{G}\}$ ,
- (ii)  $SEM(C)^{n+1} = SEM(C) \circ SEM(C)^n$ .

The infinite iteration is defined by:

$$SEM(C)^\infty = \{SEM(C) \circ SEM(C) \circ \dots\}.$$

### 5.3 Further Notions and Definitions

In this section some notions and definitions around *alap*-expressions are given.

***C*-run.** An element of the semantics of an *alap*-expression  $C$ ,  $SEM(C, G)$  respectively  $SEM(C)$ , is called a  $C_G$ -run respectively  $C$ -run.

***C*-reconcilable derivations.** A  $C$ -reconcilable derivation is a derivation which is not permitted so far (and it's prolongation perhaps never will be), but not contradicts  $C$  yet. In the following we define infinite  $C$ -reconcilable derivations.

**Definition 68.** Infinite  $C$ -reconcilable derivations for a specific input graph

- $Inf(\emptyset, G) = \emptyset$
- $Inf(\lambda, G) = \emptyset$
- $Inf(r, G) = \emptyset$
- $Inf(C_1; C_2, G) = Inf(C_1, G) \cup SEM(C_1, G) \circ \bigcup_{G \xrightarrow{*} G' \in SEM(C_1, G)} Inf(C_2, G')$
- $Inf(C_1|C_2, G) = Inf(C_1, G) \cup Inf(C_2, G)$
- $Inf(C^*, G) = Inf(C, G) \cup SEM(C, G)^\infty \cup \bigcup_{k \in \mathbb{N}} (SEM(C, G)^k \circ \bigcup_{G \xrightarrow{*} G' \in SEM(C_1, G)^k} Inf(C, G'))$
- $Inf(C!, G) = Inf(C, G) \cup SEM(C, G)^\infty \cup \bigcup_{k \in \mathbb{N}} (SEM(C, G)^k \circ \bigcup_{G \xrightarrow{*} G' \in SEM(C_1, G)^k} Inf(C, G'))$

**Definition 69.** Infinite  $C$ -reconcilable derivations for all graphs  $G \in \mathcal{G}$

- $Inf(\emptyset) = \emptyset$
- $Inf(\lambda) = \emptyset$
- $Inf(r) = \emptyset$
- $Inf(C_1; C_2) = Inf(C_1) \cup SEM(C_1) \circ Inf(C_2)$
- $Inf(C_1|C_2) = Inf(C_1) \cup Inf(C_2)$

- $\text{Inf}(C^*) = \text{Inf}(C) \cup \text{SEM}(C)^\infty \cup \bigcup_{k \in \mathbb{N}} (\text{SEM}(C)^k \circ \text{Inf}(C))$
- $\text{Inf}(C!) = \text{Inf}(C) \cup \text{SEM}(C)^\infty \cup \bigcup_{k \in \mathbb{N}} (\text{SEM}(C)^k \circ \text{Inf}(C))$

## Chapter 6

# Sufficient Conditions for Termination of As-long-as-possible Expressions

Considering mere regular expressions the question of termination only is of concern when asking if the semantics is finite or infinite. Regarding the computation of single semantic elements each running computation turns out to be permitted or not eventually. When the *alap*-operator comes into play this situation changes. Termination now is essential to answer the question if a given running computation will end up to be permitted or not.

Termination in general is an important property. On the one hand termination of the modelled processes itself is essential (at least if they are expected to do so). Consider the modelling of negotiations between trucks and packages over pick up and delivery in a logistic environment. It is crucial that the negotiations will come to an end eventually. On the other hand termination is a necessary property for verification, i.e. proving the total correctness of the developed model. Despite termination is undecidable in general, it is worthwhile to examine *alap*-expressions to the effect if they definitely terminate.

This chapter makes use of the formality of graph transformation and introduces sufficient conditions for termination.

After a short outline of the consideration of termination in literature this chapter introduces termination regarding *alap*-expressions employing Petri nets.

## 6.1 Termination in Literature

Termination in general is undecidable for most modelling frameworks (see, e.g. [Plu98]), meaning that there is no general procedure, which applied to a model of a process will tell whether the process will terminate or not. A well known textbook example in computer science, considering the undecidability of termination, is the halting problem, which can be stated as follows: given a program (which also is a process) and an input for that program, decide whether the program will eventually halt when run with that input, or will run forever. Alan Turing proved that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist. However in many practical applications one does not have to prove termination in general, but for special cases. Therefore the main approaches regarding termination specify sufficient conditions for termination for specific frameworks.

A lot of approaches base on the classical approach to guarantee termination. In this approach termination is proved by finding a valuation function that associates a natural number to each graph with the additional property that the value decreases whenever a rule application step is done. Since no natural number can decrease forever the process of rule application has to stop eventually. This valuation function then is called a termination function. Let  $P$  be a set of rules and  $val : G \rightarrow \mathbb{N}$  be a valuation function with  $val(G) > val(G')$  for each direct derivation  $G \xRightarrow{P} G'$ . Then each derivation composed of direct derivations has to be finite. Alternatively, one may replace the natural numbers by some ordered domain which does not have any decreasing infinite sequence (see, e.g., [DM78]). Unfortunately, this criterion applies only to very special processes and does not work for many others although they terminate, too. The reason is that one may not find any evaluation function that decreases in each step. In other cases there may exist such a function, but it is hard to find.

Sufficient conditions for termination were focused by e.g. [BHPT05] or [VVE<sup>+</sup>06]. In [BHPT05] sufficient termination conditions for high-level replacement units with regular expressions without the Kleene-star operator, but with as-long-as-possible were introduced. The rough idea is to extend the notion of rule application to derived rules, i.e. several sequentially applied rules are summarised to one rule, the derived rule, and find a termination function which works for this derived rule. For an expression with several sequentially composed as-long-as-possible expressions, there may be several termination functions. In [VVE<sup>+</sup>06] a sufficient termination criterion

for graph transformation systems was proposed. In that paper the authors abstracted from the structure of the graphs and applied Petri nets to simulate the graph transformation system. If the Petri net runs out of tokens sometime they can conclude that the original graph transformation system is terminating.

## 6.2 Termination Regarding *Alap*-expressions

Our approach, called *assured termination*, enhances the idea of the classical approach in order to include more terminating processes. Thereby, it needs no explicitly given termination function. In contrast to the classical approach our approach considers entire derivations as steps instead of single rule applications. Thus, not in every single rule application step there has to decrease something. The rough idea is to exploit the knowledge we have about the structure of the derivations permitted by an *alap*-expression. All permitted derivations have particular structures that are known in advance. They are applications of rule sequences and of iterated rule sequences. Non-termination of the derivation process is only caused by the iterative application of rule sequences, which only arises due to the operators  $*$  and  $!$  of the underlying expression. The application of rule sequences is finite and therefore causes no non-termination. The idea is to estimate what is deleted and added by those iterated sequences and project whether they can be repeated infinitely often or not. This projection is done with the help of Petri nets and their algebraic analysing methods.

In order to develop the notion of assured termination, in a first step we define termination with respect to *alap* expressions. Thereby we distinguish two types of termination, namely *strong* and *weak* termination. In a second step we restrict what we have defined by strong termination to a termination criterion called *structural* termination. This is done by taking into account only the structure of the as-long-as-possible expression without considering the input graph and its transformation during the derivation process. In a last step we implement the above roughly sketched approach. There we also have to make some restrictions which eventually leads to the notion of *assured* termination. The chapter concludes with an algorithm to check assured termination and an example.



## 6.2.1 Strong Termination

In this thesis the term termination refers to the computation of the semantics of a control condition. Here one can distinguish between weak termination and strong termination. Weak termination means that at least one element of the semantics can be computed in finite time, or it can be decided in finite time that there are no elements at all. Strong termination denotes that all elements of the semantics can be computed in finite time. In general weak and strong termination depend on the input graph.

The focus of this thesis is on strong termination. In the further we refer to 'strong termination of the derivation process regulated by a control condition  $C$ ' by the notion of ' $C$  terminates strongly'. As said above, informally speaking, strong termination of a control condition is that all elements of the semantics can be computed in finite time. That means during the computation of the semantics only finitely long and finitely many derivations arise. This is expressed by the following definition.

**Definition 70. Strong termination regarding a specific input graph**

Let  $C$  be an *alap* expression over  $\mathcal{R}$  and  $G \in \mathcal{G}$  a graph.  $(C, G)$  *terminates strongly* if there is no infinitely long  $(C, G)$ -reconcilable derivation, i.e.  $\text{Inf}(C, G) = \emptyset$ .

Please note, that in order to determine the deletion and addition of graph elements by rules we consider only graph transformation approaches with injective matches. Without injective matches a rule could be considered e.g. to delete two items but through identification of these two items by a non injective match actually deletes only one item.

The following Lemma 71 introduces some compositional properties of *alap* expressions regarding strong termination.

**Lemma 71. Properties of strongly terminating control conditions**

Let  $C, C_1$ , and  $C_2$  be *alap* expressions over  $\mathcal{R}$  and  $G, G' \in \mathcal{G}$ .

- $(\emptyset, G)$  terminates strongly for all  $G \in \mathcal{G}$ .
- $(\lambda, G)$  terminates strongly for all  $G \in \mathcal{G}$ .
- $(r, G)$  terminates strongly for all  $G \in \mathcal{G}$ .
- $(C_1; C_2, G)$  terminates strongly, iff  $(C_1, G)$  terminates strongly and  $\forall G' : G \xrightarrow{*} G' \in \text{SEM}(C_1, G)$   $(C_2, G')$  terminates strongly.

- $(C_1|C_2, G)$  terminates strongly iff  $(C_1, G)$  terminates strongly and  $(C_2, G)$  terminates strongly.
- $(C^*, G)$  terminates strongly iff  $(C, G)$  terminates strongly and  $SEM(C, G)^\infty = \emptyset$  and  $\forall G'$  with  $G \xrightarrow{*} G' \in SEM(C, G)^k, k \in \mathbb{N}$   $(C, G')$  terminates strongly.
- $(C!, G)$  terminates strongly iff  $(C, G)$  terminates strongly and  $SEM(C, G)^\infty = \emptyset$  and  $\forall G'$  with  $G \xrightarrow{*} G' \in SEM(C, G)^k, k \in \mathbb{N}$   $(C, G')$  terminates strongly.

*Proof.* Let  $C, C_1, C_2$  be *alap* expressions over  $\mathcal{R}$ .

- $(\emptyset, G)$  terminates strongly for all  $G \in \mathcal{G} \Leftrightarrow Inf(\emptyset, G) = \emptyset$  for all  $G \in \mathcal{G}$ . This is true by definition.
- $(\lambda, G)$  terminates strongly for all  $G \in \mathcal{G} \Leftrightarrow Inf(\lambda, G) = \emptyset$  for all  $G \in \mathcal{G}$ . This is true by definition.
- $(r, G)$  terminates strongly for all  $G \in \mathcal{G} \Leftrightarrow Inf(r, G) = \emptyset$  for all  $G \in \mathcal{G}$ . This is true by definition.
- $(C_1; C_2, G)$  terminates strongly  $\Leftrightarrow Inf(C_1; C_2, G) = \emptyset$   
 $\Leftrightarrow Inf(C_1, G) \cup SEM(C_1, G) \circ \bigcup_{G \xrightarrow{*} G' \in SEM(C_1, G)} Inf(C_2, G') = \emptyset$   
 $\Leftrightarrow Inf(C_1, G) = \emptyset$  and  $SEM(C_1, G) \circ \bigcup_{G \xrightarrow{*} G' \in SEM(C_1, G)} Inf(C_2, G') = \emptyset$   
 $\Leftrightarrow Inf(C_1, G) = \emptyset$  and  $\bigcup_{G \xrightarrow{*} G' \in SEM(C_1, G)} Inf(C_2, G') = \emptyset$   
 $\Leftrightarrow (C_1, G)$  terminates strongly and  $(C_2, G')$  terminates strongly  $\forall G \xrightarrow{*} G' \in SEM(C_1, G)$
- $(C_1|C_2, G)$  terminates strongly  $\Leftrightarrow Inf(C_1|C_2, G) = \emptyset$   
 $\Leftrightarrow Inf(C_1, G) \cup Inf(C_2, G) = \emptyset$   
 $\Leftrightarrow Inf(C_1, G) = \emptyset$  and  $Inf(C_2, G) = \emptyset$   
 $\Leftrightarrow (C_1, G)$  and  $(C_2, G)$  terminate strongly
- $(C^*, G)$  terminates strongly  $\Leftrightarrow Inf(C^*, G) = \emptyset$   
 $\Leftrightarrow Inf(C, G) \cup SEM(C, G)^\infty \cup \bigcup_{k \in \mathbb{N}} (SEM(C, G)^k \circ \bigcup_{G \xrightarrow{*} G' \in SEM(C, G)^k} Inf(C, G')) = \emptyset$   
 $\Leftrightarrow Inf(C, G) = \emptyset$  and  $SEM(C, G)^\infty = \emptyset$  and  
 $SEM(C, G)^k \circ \bigcup_{G \xrightarrow{*} G' \in SEM(C, G)^k} Inf(C, G') = \emptyset, k \in \mathbb{N}$   
 $\Leftrightarrow (C, G)$  terminates strongly and  $SEM(C, G)^\infty = \emptyset$  and

$$\bigcup_{G \xrightarrow{*} G' \in SEM(C, G)^k} Inf(C, G') = \emptyset, k \in \mathbb{N} \quad [\text{def. 70}]$$

$$\Leftrightarrow (C, G) \text{ terminates strongly and } SEM(C, G)^\infty = \emptyset \text{ and}$$

$$(C, G') \text{ terminates strongly } \forall G \xrightarrow{*} G' \in SEM(C, G)^k, k \in \mathbb{N}$$

- The case  $C!$  works analogous to  $C^*$

□

### Definition 72. Strong termination for all input graphs

Let  $C$  be an *alap* expression over  $\mathcal{R}$ .

$C$  terminates strongly if  $(C, G)$  terminates strongly for all  $G \in \mathcal{G}$ .

**Remark 1.** Note that in the case of strong termination for all input graphs, despite termination of all the single derivation processes, the checking will not terminate if there are infinitely many input graphs. This problem is taken up in Section 6.2.3.

Lemma 71 indicates why termination is undecidable in general. In order to decide strong termination for an as-long-as-possible expression one has to compute all the intermediate graphs of its described derivations, i.e. all the possible derivations. If this were realisable we would not have to ask the question of termination. To cope with this problem we have to restrict the demands. This leads to the notion of *structural* termination, which is focused in the following section.

## 6.2.2 Structural Termination

As we cannot compute all the intermediate graphs structural termination of an as-long-as-possible expression  $C$  demands that all subexpressions of  $C$  must terminate strongly for all graphs, i.e. strong termination is inherent in the structure of the expression.

### Definition 73. Structural termination

Let  $C$  be an *alap* expression over  $\mathcal{R}$ .

$C$  terminates structurally if all subexpressions of  $C$  terminate strongly for all graphs  $G \in \mathcal{G}$ .

**Remark 2.** Structural termination implies strong termination, as each expression is a subexpression of itself.

### Lemma 74. Properties of structurally terminating control conditions

Let  $C_0, C_1, C_2$  be *alap* expressions over  $\mathcal{R}$ .

- $\emptyset$  terminates structurally.
- $\lambda$  terminates structurally.
- $r$  terminates structurally.
- $C_1; C_2$  terminates structurally iff  $C_1$  terminates structurally and  $C_2$  terminates structurally.
- $C_1|C_2$  terminates structurally iff  $C_1$  terminates structurally and  $C_2$  terminates structurally.
- $C_0^*$  terminates structurally iff  $C_0$  terminates structurally and  $SEM(C_0)^\infty = \emptyset$ .
- $C_0!$  terminates structurally iff  $C_0$  terminates structurally and  $SEM(C_0)^\infty = \emptyset$ .

*Proof.* Let  $C, C_1, C_2$  be control conditions over  $\mathcal{R}$ . Assumption: every expression  $C_i$  is considered to be parenthesised.

- $\emptyset$  terminates structurally  $\Leftrightarrow \emptyset$  terminates strongly for all graphs. This is true by definition.
- $\lambda$  terminates structurally  $\Leftrightarrow \lambda$  terminates strongly for all graphs. This is true by definition.
- $r$  terminates structurally  $\Leftrightarrow r$  terminates strongly for all graphs. This is true by definition.
- $C_1; C_2$  terminates structurally
  - $\Leftrightarrow$  All subexpressions of  $C_1; C_2$  terminate strongly for all graphs
  - $\Leftrightarrow$  All subexpressions of  $C_1$  terminate strongly for all graphs and all subexpressions of  $C_2$  terminate strongly for all graphs
  - $\Leftrightarrow C_1$  terminates structurally and  $C_2$  terminates structurally
- $C_1|C_2$  terminates structurally
  - $\Leftrightarrow$  All subexpressions of  $C_1|C_2$  terminate strongly for all graphs
  - $\Leftrightarrow$  All subexpressions of  $C_1$  terminate strongly for all graphs and all subexpressions of  $C_2$  terminate strongly for all graphs
  - $\Leftrightarrow C_1$  terminates structurally and  $C_2$  terminates structurally
- $C^*$  terminates structurally
  - $\Leftrightarrow$  All subexpressions of  $C^*$  terminate strongly for all graphs [def. 73]
  - $\Leftrightarrow C^*$  terminates strongly for all graphs and all subexpressions of  $C$

terminate strongly for all graphs  
 $\Leftrightarrow C$  terminates strongly for all graphs and  $SEM(C)^\infty = \emptyset$  and  $\forall G \xRightarrow{*} G' \in SEM(C)^k, k \in \mathbb{N}(C, G')$  terminates strongly and all subexpressions of  $C$  terminate strongly for all graphs [prop.71]  
 $\Leftrightarrow C$  terminates strongly for all graphs and  $SEM(C)^\infty = \emptyset$  and all subexpressions of  $C$  terminate strongly for all graphs  
 $\Leftrightarrow C$  terminates strongly for all graphs and all subexpressions of  $C$  terminate strongly for all graphs and  $SEM(C)^\infty = \emptyset$   
 $\Leftrightarrow C$  terminates structurally and  $SEM(C)^\infty = \emptyset$

- The case  $C!$  works analogous to  $C^*$ .

□

Non-termination of an as-long-as-possible expression  $C$  is only caused by  $*$  and  $!$ . The following lemma reduces the definition of structural termination to statements about subexpressions of the form  $C_0^*$  and  $C_0!$ .

**Lemma 75.**  $C$  terminates structurally if and only if for all subexpressions  $C_0^*$  and  $C_0!$  of  $C$  holds  $SEM(C_0)^\infty = \emptyset$ .

*Proof.* Let  $C$  be a control condition over  $\mathcal{R}$ .

We show the proposition by induction over the structure of  $C$ .

*Basis:* For  $C = \emptyset, \lambda, r$  the hypothesis is true, since  $\emptyset, \lambda, r$  terminate structurally and contain no subexpression  $C_0^*$  or  $C_0!$ .

*Step:* Examine each of the cases (i)  $C = C_1; C_2$ , (ii)  $C = C_1|C_2$ , (iii)  $C = C'^*$ , (iv)  $C = C'!$  separately. In order to keep the proof overseeable we abbreviate the term subexpressions by sbxps.

(i)  $C = C_1; C_2$   
 $C_1; C_2$  terminates structurally  
 $\Leftrightarrow C_1$  terminates structurally and  $C_2$  terminates structurally [lemma 74]  
 $\Leftrightarrow \forall$  sbxps of  $C_1$  and  $C_2$  of the form  $C_0^*$  and  $C_0!$  holds  $SEM(C_0)^\infty = \emptyset$  [i.h]  
 $\Leftrightarrow \forall$  sbxps  $C_0^*$  and  $C_0!$  of  $C_1; C_2$  holds  $SEM(C_0)^\infty = \emptyset$

(ii)  $C = C_1|C_2$   
 $C_1|C_2$  terminates structurally  
 $\Leftrightarrow C_1$  terminates structurally and  $C_2$  terminates structurally [lemma 74]  
 $\Leftrightarrow \forall$  sbxps of  $C_1$  and  $C_2$  of the form  $C_0^*$  and  $C_0!$  holds  $SEM(C_0)^\infty = \emptyset$  [i.h]  
 $\Leftrightarrow \forall$  sbxps  $C_0^*$  and  $C_0!$  of  $C_1|C_2$  holds  $SEM(C_0)^\infty = \emptyset$

(iii)  $C = C'^*$   
 $C'^*$  terminates structurally  
 $\Leftrightarrow C'$  terminates structurally and  $SEM(C')^\infty = \emptyset$  [lemma 74]  
 $\Leftrightarrow (\forall \text{ sbxps } C_0^* (C_0!) \text{ of } C' SEM(C_0)^\infty = \emptyset)$  and  $SEM(C')^\infty = \emptyset$  [i.h.]  
 $\Leftrightarrow \forall \text{ sbxps } C_0^* (C_0!) \text{ of } C'^* SEM(C_0)^\infty = \emptyset$

(iv)  $C = C'!$  works analogously to (iii). □

### 6.2.3 Assured Termination

According to Lemma 75 in order to decide if  $C$  terminates structurally we have to decide whether  $SEM(C_0)^\infty = \emptyset$  for all subexpressions  $C_0$  of an as-long-as-possible expression  $C$ . It holds that  $SEM(C_0)^\infty = \emptyset$  if there exists no infinitely long derivation build by concatenations of  $C_0$ -runs. Such an infinitely long derivation only exists if again and again after an execution of a  $C_0$ -run a further entire  $C_0$ -run is executable. This is only the case if the preconditions for the execution are complied, i.e. every graph structure that is needed by  $C_0$  is provided by the current input graph. This is only the case if the respective structures are not deleted during execution of the  $C_0$ -runs or they are deleted, but also added again in a sufficient amount.

#### Informal description of the idea:

$SEM(C)^\infty = \emptyset$  if

1. Every  $C$ -run deletes at least one graph item and
2. during the interaction of the different  $C$ -runs, at least one deleted graph item of every  $C$ -run cannot eventually be counterbalanced by another  $C$ -run adding the same item.

With this approach we are able to make statements about sufficient conditions for termination without having to examine all possible input graphs. In order to formalise the informal description we define measures on graphs to represent what can be deleted from a graph respectively added, and define means to estimate upper bounds for the changes on these measures inflicted by runs of a control condition  $C$ . After that we model the defined structures with Petri nets and use their algebraic analysing methods to investigate the possibilities of interaction between the different  $C$ -runs.

## Measure sets for graphs

A measure maps graphs to natural numbers such that for each rule its application yields the same change in the measured value, independently from the graph to which the rule is applied.

### Definition 76. Measure on graphs

A *measure* on graphs is a mapping  $\mu: \mathcal{G} \rightarrow \mathbb{N}$  such that for all graphs  $G, G', \overline{G}, \overline{G}' \in \mathcal{G}$  and every rule  $r \in \mathcal{R}$ ,  $G \xRightarrow[r]{} G'$  and  $\overline{G} \xRightarrow[r]{} \overline{G}'$  implies  $\mu(G') - \mu(G) = \mu(\overline{G}') - \mu(\overline{G})$ . We will write a set of  $k$  measures  $\mu_1, \dots, \mu_k$  ( $k \in \mathbb{N}$ ) as a vector  $\vec{\mu} = (\mu_i)_{i \in [k]}$ .

Possible examples for measures are: number of nodes, number of edges, number of  $a$ -labelled edges or number of  $b$ -labelled loops for some symbols  $a, b$  of the graph-labelling alphabet. Of course, which mappings qualify as measures depends on the graph transformation approach and the rules occurring in the considered control condition. For instance, node-rewriting rules (see, e.g., [ER97]) usually admit implicit multiplication of embedding edges, so that an edge-based mapping is no measure. If, however, every rule in the concrete set will just transfer every incident  $a$ -labelled edge to exactly one replacing node, counting  $a$ -labelled edges is valid as a measure. As said above we employ a graph transformation approach with injective matches, so we are able to use every graph item and label as basis for a measure.

## Change(C) – Estimate upper bounds for C-runs

In order to represent the changes in graph measures inflicted by execution of a control condition  $C$ , we define a set  $Change(C)$ , which contains for every  $C$ -run a vector indicating the changes inflicted by the  $C$ -run. Since there could be infinitely many or infinitely long  $C$ -runs it is not possible to have one change-vector for every  $C$ -run which represents the exact changes of the respective run. The changes have to be estimated under worst case assumptions. This is the reason for the notion of *assured termination*. We define a set  $Change(C) \subseteq \mathbb{Z}_\infty^k$  of vectors for each control condition  $C$  so that each vector has  $k$  entries in  $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$  that serve as upper bounds for the change in measured values whenever a derivation admitted by  $C$  is executed.

### Definition 77. Change(C)

Let  $C$  be an *alap* expression over  $\mathcal{R}$  and  $\vec{\mu}$  a measure set.

1. For  $C = \emptyset, \lambda$ , let  $Change(C) = \emptyset$

2. For  $C = r \in \mathcal{R}$ , let  $Change(C) = \{x\}$ , where  $x$  is the unique vector  $x = \vec{\mu}(G') - \vec{\mu}(G)$  for all  $G, G' \in \mathcal{G}$  with  $G \Rightarrow_r G'$ , and vector difference is computed component-wise.
3. For  $C = C_1; C_2$ , let  $Change(C) = \{x + y \mid x \in Change(C_1), y \in Change(C_2)\}$ , where vector addition is computed component-wise.
4. For  $C = C_1|C_2$ , let  $Change(C) = Change(C_1) \cup Change(C_2)$ .
5. For  $C = C_0^*$ ,  $C = C_0!$  let  $Change(C) = \{(x_1, \dots, x_k) \mid x_i = \infty \text{ if } \exists (y_1, \dots, y_k) \in Change(C_0) : y_i > 0 \text{ and } x_i = 0 \text{ otherwise, for } i \in [k]\}$ .

The reasons for the entries of  $Change(C)$  in item 5. are the following: for an expression  $C_0$  that terminates strongly,  $C_0^*$  admits rule application sequences where  $C_0$  is iterated arbitrarily often, and  $C_0!$  where  $C_0$  is iterated as long as possible. Any decrease in a measure through a  $C_0$ -run does not occur if  $C_0$  is iterated zero times. In contrast, an increase in a measure may lead to arbitrarily large values of that measure, indicated by  $\infty$ .

The following lemma states that all changes in the values of measures of graphs transformed by the application of an *alap*-expression  $C$  are bounded by  $Change(C)$ .

**Lemma 78.** Let  $C$  be an *alap* expression over  $\mathcal{R}$ .

a) Finite case:

For all  $G, G' \in \mathcal{G}$  with  $G \xRightarrow{*} G' \in SEM(C)$  there exists  $x \in Change(C)$  with  $\vec{\mu}(G') - \vec{\mu}(G) \leq x$ .

b) Infinite case:

For all  $G \xRightarrow{*} G_{i_1} \xRightarrow{*} G_{i_2} \xRightarrow{*} \dots \in SEM(C)$  there exist a strictly increasing sequence  $(p_1, p_2, \dots), p_k \in \mathbb{N}$  and  $x \in Change(C)$  such that  $\vec{\mu}(G_{p_k}) - \vec{\mu}(G) \leq x$ , for all  $k \in \mathbb{N}$ .

*Proof.* We show the assertions by induction over the structure of  $C$ .

**a) Finite case:**

*Basis:* For  $C = \emptyset$  or  $C = \lambda$ , the assertion is true, since  $SEM(\emptyset)$  and  $SEM(\lambda)$  contain no elements.

For  $C = r \in \mathcal{R}$ , the assertion follows from the definition, i.e.  $Change(r) = \{x\}$  with  $x = \vec{\mu}(G') - \vec{\mu}(G)$  for all  $G, G' \in \mathcal{G}$  with  $G \Rightarrow_r G'$ .

*Step:* Examine each of the cases (i)  $C = C_1; C_2$ , (ii)  $C = C_1|C_2$ , (iii)  $C = C_0^*(!)$  separately.

- (i) Let  $C = C_1; C_2$  with  $G \xRightarrow{*} \overline{G} \xRightarrow{*} G' \in SEM(C)$ , and  $G \xRightarrow{*} \overline{G} \in SEM(C_1), \overline{G} \xRightarrow{*} G' \in SEM(C_2)$



By induction hypothesis, we have

$$\vec{\mu}(\overline{G}) - \vec{\mu}(G) \leq x_1 \text{ for some } x_1 \in \text{Change}(C_1) \text{ and}$$

$$\vec{\mu}(G') - \vec{\mu}(\overline{G}) \leq x_2 \text{ for some } x_2 \in \text{Change}(C_2).$$

$$\begin{aligned} \text{Consequently, } \vec{\mu}(G') - \vec{\mu}(G) &= \vec{\mu}(G') + (-\vec{\mu}(\overline{G}) + \vec{\mu}(\overline{G})) - \vec{\mu}(G) = \\ &= (\vec{\mu}(G') - \vec{\mu}(\overline{G})) + (\vec{\mu}(\overline{G}) - \vec{\mu}(G)) \leq x_2 + x_1 = x_1 + x_2. \end{aligned}$$

$x_1 + x_2 \in \text{Change}(C_1; C_2)$  by definition.

(ii) Let  $C = C_1|C_2$  and  $G \xrightarrow{*} G' \in \text{SEM}(C)$ . If  $G \xrightarrow{*} G' \in \text{SEM}(C_1)$ , then by induction hypothesis there exists  $x_1 \in \text{Change}(C_1)$  with  $\vec{\mu}(G') - \vec{\mu}(G) \leq x_1$ . Moreover,  $\text{Change}(C) = \text{Change}(C_1) \cup \text{Change}(C_2)$ , so that  $x_1 \in \text{Change}(C)$ . The case  $G \xrightarrow{*} G' \in \text{SEM}(C_2)$  is analogous.

(iii) Let  $C = C_0^*(!)$ ,  $G \xrightarrow{*} G' \in \text{SEM}(C)$ , and  $\{x\} = \text{Change}(C)$ .

Since  $G \xrightarrow{*} G' \in \text{SEM}(C) = \text{SEM}(C_0^*(!))$  there are  $n \in \mathbb{N}$  and  $(G = G_0) \xrightarrow{*} G_1, G_1 \xrightarrow{*} G_2, \dots, G_{n-1} \xrightarrow{*} (G_n = G')$  with  $G_{i-1} \xrightarrow{*} G_i \in \text{SEM}(C_0), i \in \{1, \dots, n\}$  and  $(G = G_0) \xrightarrow{*} G_1 \xrightarrow{*} G_2 \xrightarrow{*} \dots \xrightarrow{*} (G_n = G') \in \text{SEM}(C)$ .

By induction hypothesis there exists  $x_1, \dots, x_n \in \text{Change}(C_0)$  so that  $\vec{\mu}(G_i) - \vec{\mu}(G_{i-1}) \leq x_i$  for all  $i \in \{1, \dots, n\}$ . Similarly to (i) we obtain:  $\vec{\mu}(G') - \vec{\mu}(G) = \vec{\mu}(G_n) - \vec{\mu}(G_0) = \sum_{i=1}^n \vec{\mu}(G_i) - \vec{\mu}(G_{i-1}) \leq \sum_{i=1}^n x_i =: x_0$ . In order to prove that  $x_0 \leq x$ , we have to compare their entries. If the  $k$ -th entry of  $x_0$  is greater than 0, there is at least one  $x_i$  whose  $k$ -th entry is also greater than 0, so that the  $k$ -th entry of  $x$  is  $\infty$ . Otherwise, the  $k$ -th entry of  $x_0$  is at most 0, which is the minimal entry that  $x$  may have.

### b) Infinite case:

Basis: Let  $C = \emptyset, \lambda, r$ . There exists no  $G \xrightarrow{*} G_{i_1} \xrightarrow{*} G_{i_2} \xrightarrow{*} \dots \in \text{SEM}(C)$ , hence the proposition is true.

Inductive Step: Examine each of the cases (i)  $C = C_1; C_2$ , (ii)  $C = C_1|C_2$ , (iii)  $C = C_0^*$ , (iv)  $C = C_0!$  separately.

(i) Let  $C = C_1; C_2$  and  $G = G_0 \xrightarrow{*} G_n \xrightarrow{*} G_{n+1} \xrightarrow{*} \dots \in \text{SEM}(C_1; C_2)$  with  $G \xrightarrow{*} G_n \in \text{SEM}(C_1)$  and  $G_n \xrightarrow{*} G_{n+1} \xrightarrow{*} \dots \in \text{SEM}(C_2)$ . By a) there exist  $x_1 \in \text{Change}(C_1)$  with  $\vec{\mu}(G_n) - \vec{\mu}(G) \leq x_1$  and by induction hypothesis there exists a strictly increasing sequence  $(j_1, j_2, \dots), j_k \in \mathbb{N}$  as well as  $x_2 \in \text{Change}(C_2)$  with  $\vec{\mu}(G_{j_k}) - \vec{\mu}(G_n) \leq x_2$ .

Let the sequence  $(p_1, p_2, \dots) = (j_1, j_2, \dots)$  and  $x = x_1 + x_2 \in \text{Change}(C_1; C_2)$ .

To show:  $\vec{\mu}(G_{p_k}) = \vec{\mu}(G_{j_k}) - \vec{\mu}(G) \leq x$ .  
 $\vec{\mu}(G_n) - \vec{\mu}(G) + \vec{\mu}(G_{j_k}) - \vec{\mu}(G_n) \leq x_1 + x_2 \Leftrightarrow$   
 $\vec{\mu}(G_{j_k}) - \vec{\mu}(G) \leq x_1 + x_2 \Leftrightarrow$   
 $\vec{\mu}(G_{p_k}) - \vec{\mu}(G) \leq x_1 + x_2 = x$

(ii) Let  $C = C_1|C_2$ .

Let  $G_0 \xrightarrow{*} G_1 \xrightarrow{*} G_2 \xrightarrow{*} \dots \in SEM(C_1|C_2)$ . If  $G_0 \xrightarrow{*} G_1 \xrightarrow{*} G_2 \xrightarrow{*} \dots \in SEM(C_1)$ , then by induction hypothesis there exists  $x_1 \in Change(C_1)$  and a strictly increasing sequence  $(j_1, j_2, \dots), j_k \in \mathbb{N}$  with  $\vec{\mu}(G_{j_k}) - \vec{\mu}(G) \leq x_1$ . Moreover,  $Change(C) = Change(C_1) \cup Change(C_2)$ , such that  $x_1 \in Change(C)$ . The case  $G_0 \xrightarrow{*} G_1 \xrightarrow{*} G_2 \xrightarrow{*} \dots \in SEM(C_2)$  works analogous.

(iii) Let  $C = C_0^*$  with  $G \xrightarrow{*} G_1 \dots \xrightarrow{*} G_n \xrightarrow{*} G_{n+1} \xrightarrow{*} \dots \in SEM(C_0^*)$ ,  $n \in \mathbb{N}$ , with  $(G = G_0) \xrightarrow{*} G_n \in SEM(C_0)^n$ ,  $G_{i-1} \xrightarrow{*} G_i \in SEM(C_0), i \in [n]$  and  $G_n \xrightarrow{*} G_{n+1} \xrightarrow{*} \dots \in SEM(C_0)$ . Moreover let  $\{x\} = Change(C_0^*)$ . To show: There exists a strictly increasing sequence  $(p_1, p_2, \dots), p_k \in \mathbb{N}$  with  $\vec{\mu}(G_{p_k}) - \vec{\mu}(G) \leq x$ .

By a) there exist  $x_1, \dots, x_n \in Change(C_0)$  with  $\vec{\mu}(G_i) - \vec{\mu}(G_{i-1}) \leq x_i$  for all  $i \in \{1, \dots, n\}$ . Moreover by induction hypothesis there exist  $x' \in Change(C_0)$  and a strictly increasing sequence  $(j_1, j_2, \dots)_{j_k \in \mathbb{N}, j_k > n}$  with  $\vec{\mu}(G_{j_k}) - \vec{\mu}(G_n) \leq x'$ .

Let  $(p_1, p_2, \dots) = (j_1, j_2, \dots)$ .  $\vec{\mu}(G_1) - \vec{\mu}(G_0) + \vec{\mu}(G_2) - \vec{\mu}(G_1) + \dots + \vec{\mu}(G_n) - \vec{\mu}(G_{n-1}) + \vec{\mu}(G_{p_k}) - \vec{\mu}(G_n) \leq \sum_{i \in [n]} x_i + x' \Leftrightarrow$

$$\vec{\mu}(G_{p_k}) - \vec{\mu}(G_0) \leq \sum_{i \in [n]} x_i + x'$$

Let  $x_0 := \sum_{i \in [n]} x_i$  To show:  $x_0 + x' \leq x$ . In order to prove that  $x_0 + x' \leq x$ ,

we have to compare the entries of  $x_0 + x'$  and  $x$ . If the k-th entry of  $x_0 + x'$  is greater than 0, then the k-th entry of  $x_0$  or of  $x'$  is greater than 0. If the k-th entry of  $x_0$  is greater than 0, there is at least one  $x_i$  whose k-th entry is also greater than 0, so that the k-th entry of  $x$  is  $\infty$ . A similar argument applies to  $x'$ . If the the k-th entry of  $x'$  is greater than 0, the k-th entry of  $x$  is  $\infty$ . Otherwise, the k-th entry of  $x_0 + x'$  is at most 0, which is the minimal entry that  $x$  may have.

(iv) Let  $C = C_0!$  with  $d = (G = G_0) \xrightarrow{*} G_1 \xrightarrow{*} G_2 \xrightarrow{*} \dots \in SEM(C_0!)$  with

- 1)  $d \in SEM(C_0)^\infty$  and  $G_i \xrightarrow{*} G_{i+1} \in SEM(C_0)$  for  $i \in \mathbb{N}$ , or
- 2)  $d_1 = (G = G_0) \xrightarrow{*} G_n \in SEM(C_0)^n$  for  $n \in \mathbb{N}$  with  $G_i \xrightarrow{*} G_{i+1} \in$

$SEM(C_0)$  for  $i \in \mathbb{N}$  and  $d_2 = G_n \xrightarrow{*} G_{n+1} \xrightarrow{*} \dots \in SEM(C_0)$ ,  $d_1 \circ d_2 = d$ .  
 Moreover let  $\{x\} = Change(C)$ .

To show:  $\exists(p_1, p_2, \dots), p_k \in \mathbb{N}$  with  $\vec{\mu}(G_{p_k}) - \vec{\mu}(G) \leq x$

ad 1) By a) we have  $x_i \in Change(C_0)$  with  $\vec{\mu}(G_{i+1}) - \vec{\mu}(G_i) \leq x_i, i \in \mathbb{N}$ . Let  $(p_1, p_2, \dots) = (0, 1, 2, \dots)$ .

$$\vec{\mu}(G_1) - \vec{\mu}(G_0) + \vec{\mu}(G_2) - \vec{\mu}(G_1) + \dots + \vec{\mu}(G_{p_k}) - \vec{\mu}(G_{p_k-1}) \leq \sum_{i=0, \dots, p_k} x_i \Leftrightarrow$$

$$\vec{\mu}(G_{p_k}) - \vec{\mu}(G_0) \leq \sum_{i=0, \dots, p_k} x_i \Leftrightarrow$$

To show:  $\sum_{i=0, \dots, p_k} x_i \leq x$ . Like in (iii) if the k-th entry of  $\sum_{i=0, \dots, p_k} x_i$  is greater than 0, there is at least one  $x_i$  whose k-th entry is also greater than 0, so that the k-th entry of  $x$  is  $\infty$ . Otherwise, the k-th entry of  $\sum_{i=0, \dots, p_k} x_i$  is at most 0, which is the minimal entry that  $x$  may have.

ad 2) analogously to (iii) □

**Lemma 79.** Let  $C$  be an *alap* expression over  $\mathcal{R}$ .  
 $Change(C)$  is finite.

*Proof.* Induction over the structure of  $C$

*Basis:* For  $C = r \in \mathcal{R}$ ,  $Change(r)$  consists of a unique vector by definition.

*Step:*

(i) Let  $C = C_1; C_2$ . By induction hypothesis,  $Change(C_1)$  and  $Change(C_2)$  are finite. By construction,  $Change(C_1; C_2)$  contains at most the product of the sizes of  $Change(C_1)$  and  $Change(C_2)$  and therefore is finite, too.

(ii) Let  $C = C_1|C_2$ . By induction hypothesis,  $Change(C_1)$  and  $Change(C_2)$  are finite. By construction,  $Change(C_1|C_2)$  is the union of these sets and consequently finite.

(iii) Let  $C = C_0^*(!)$ . By construction,  $Change(C)$  consists of a unique vector. □

### Petri nets to model and analyse the interaction of $C$ -runs

With  $Change(C)$  we have a means to estimate what is deleted and added by  $C$ -runs. In order to decide whether  $SEM(C)^\infty = \emptyset$  we have to analyse the possibilities of interaction between the different  $C$ -runs. This is done with the help of Petri nets and their algebraic analysing methods.

Using the vectors of  $Change(C)$  we construct a Petri net. The places represent the different graph measures and the transitions represent the  $C$ -runs

(respectively subsumptions of them). The edges are marked with values of the vectors from  $Change(C)$ , so when a transition fires the graph items modified by the respective  $C$ -run are added to and subtracted from the respective places. The incidence matrix of the Petri net is given by the vectors of  $Change(C)$  as columns. Since  $Change(C)$  contains  $\infty$ -labels they also occur in the Petri net. As  $\infty$  is no number usually Petri nets do not contain  $\infty$  markings and also there are no  $\infty$  entries in a matrix, but in our case the  $\infty$  entries do no harm. Why this is the case is explained later.

**Definition 80. Constructing Petri-net  $N_C$  from  $alap$ -expression  $C$**

Let  $C$  be an  $alap$  expression over  $\mathcal{R}$  and let  $\vec{\mu} = (\mu_i)_{i \in [k]}$  be a measure set. Construct the pure Petri net  $N_C$  as follows.

- The set of places is  $P_C = \{l \in [k]\}$ ,
- the set of transitions is  $T_C = Change(C)$ ,
- the incidence matrix  $A_C$  has as columns the vectors in  $Change(C)$ .

In order to decide if there can be an infinitely long concatenation of  $C$ -runs, we have to look if there is at least one transition in the Petri net, which can fire infinitely often. This property is called partial repetitiveness and was introduced in the Preliminaries. Partial repetitiveness means that there exists an initial marking and a firing sequence such that some transitions fire infinitely often. The other way round if the Petri net is not partially repetitive for all initial marking there is no transition which can fire infinitely often hence there can not be an infinitely long sequence build by concatenations of  $C$ -runs. This is formalised by the following lemma.

**Lemma 81.** Let  $C$  be an  $alap$ -expressions and  $N_C$  the pure Petri net constructed from  $C$ . Then

$$N_C \text{ is not partially repetitive} \Rightarrow SEM(C)^\infty = \emptyset.$$

*Proof.* By contra-position to Definition 33 of partial repetitiveness we obtain:  $N_C$  is not partially repetitive  $\Rightarrow \neg(\exists M_0$  and a firing sequence  $\sigma$  from  $M_0$  such that some transition occur infinitely often in  $\sigma)$

$\Leftrightarrow N_C$  is not partially repetitive  $\Rightarrow \forall M_0$  and firing sequences  $\sigma$  from  $M_0$  all transitions occur finitely often in  $\sigma$ .

This in turn implies that there exists no infinitely long derivation  $SEM(C) \circ SEM(C) \circ \dots$ , consisting of  $C$ -runs, since all elements of  $SEM(C)$  are represented by transitions of the Petri net and there are only finitely many transitions. Therefore  $SEM(C)^\infty = \emptyset$ .  $\square$

With the help of Lemma 81 and 75 we define assured termination.

**Definition 82. Assured termination**

Let  $C$  be an *alap* expression over  $\mathcal{R}$ .

$C$  terminates assuredly if for all subexpressions  $C_0^*$  and  $C_0!$  holds  $N_{C_0}$  is not partially repetitive.

**Remark 3.** Assured termination implies structural termination.

Partial repetitiveness can be shown with the help of the incidence matrix. The changes in the graph measures caused by a derivation represented by a Parikh-vector can be computed by multiplying the incidence matrix with the Parikh-vector. When there is no Parikh-vector  $\neq 0$ , i.e. at least one  $C$ -run is executed, such that all caused changes in graph measures are non-negative, this implies that every possible combination of  $C$ -runs deletes something, i.e. there could not exist an infinitely long derivation composed of  $C$ -runs.

**Lemma 83.**  $N_C$  is not partially repetitive if and only if there is no vector  $x : T_C \rightarrow \mathbb{Z}$  of non-negative integers such that  $A_C * x \geq 0$  and  $x \neq 0$ .

*Proof.* By contra position to theorem 34 in both directions we obtain the proposition.

" $\Rightarrow$ ":  $\neg$ (there exists a  $|T|$ -vector  $x : T \rightarrow \mathbb{Z}$  of non-negative integers such that  $A * x \geq 0$  and  $x \neq 0$ )  $\Rightarrow \neg$ ( $N$  is partially repetitive)  $\Leftrightarrow$  there exists no  $|T|$ -vector  $x : T \rightarrow \mathbb{Z}$  of non-negative integers such that  $A * x \geq 0$  and  $x \neq 0 \Rightarrow N$  is not partially repetitive

" $\Leftarrow$ ": analogous □

In order to show that there is no vector  $x : T \rightarrow \mathbb{Z}$  of non-negative integers such that  $A * x \geq 0$  and  $x \neq 0$  we can use e.g. the Fourier-Motzkin elimination presented in the Preliminaries.

**Dealing with  $\infty$ -entries**

The Fourier-Motzkin elimination is not defined for entries with the value  $\infty$ , but in our case  $\infty$  does not cause trouble. When a control condition terminates assuredly it is due to other entries than those with the coefficient  $\infty$ . In order to be able to calculate, the  $\infty$  entries are replaced by variables. The inequations then of course are not resolved to this variables.

## 6.3 An Algorithm to Check Assured Termination

Using the results of this chapter we can state the following algorithm to check assured termination for an *alap* expression  $C$ .

```

CHECK( $C$  : alap expression over  $\mathcal{R}$ ) : {true, false};
case  $C \in \mathcal{R}$  :
    return true;
 $C = C_1; C_2$  or  $C = C_1|C_2$  :
    return CHECK( $C_1$ ) and CHECK( $C_2$ );
 $C = C_0^*$  or  $C = C_0!$  :
    return CHECK( $C_0$ ) and  $N_{C_0}$  is not partially repetitive
endcase

```

### 6.3.1 Example

As a simple example to show the application of the CHECK-algorithm we present the transformation unit *ShortDist*. *ShortDist* computes the shortest distance between every two nodes in a graph. It is depicted in Figure 6.3.1 and works as follows: As input it receives a "road map graph" with distance-edges (d-edges) between some nodes, representing the roads, labelled with the distance between the two respective nodes and with test-edges (t-edge) between all nodes labelled with  $t : \infty$ .

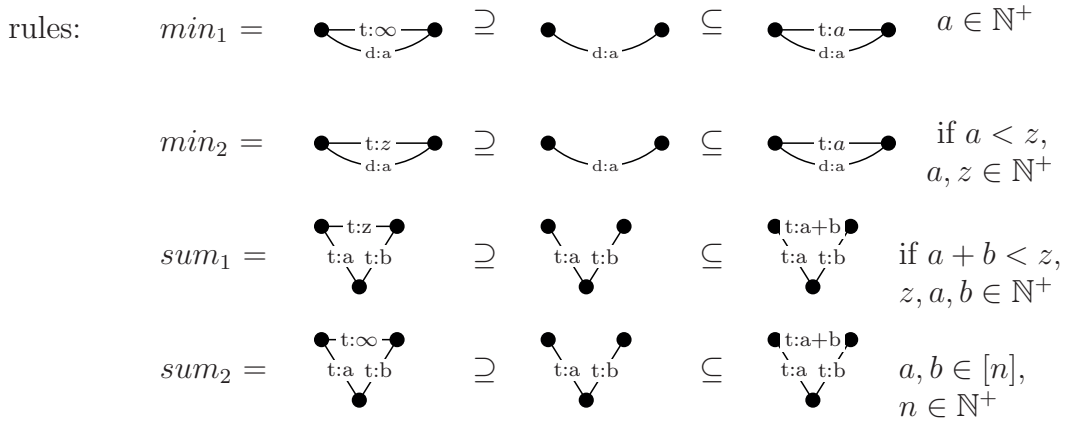
With the rules  $min_1$  and  $min_2$  applied in choice as-long-as-possible the test-edges between every two nodes are marked with the minimum of all parallel d-edges existing between these nodes, i.e. the distance of the shortest road between this nodes. ( $min_1$  relabels every t-edge, labelled with  $\infty$  with the value of an arbitrary parallel d-edge.  $min_2$  relabels an t-edge, labelled with a natural number, with the value of a parallel d-edge if this value is smaller.) The t-edges between nodes with no d-edges inbetween remain labelled with  $\infty$ .

The rules  $sum_1$  and  $sum_2$ , applied choicewise as-long-as-possible, label step by step the t-edges with the shortest distances between their nodes. The rule  $sum_1$  searches for triangles of t-edges labelled only with natural number. It relabels an edge if its label is greater than the sum of the two other edges. The rule  $sum_2$  searches for triangles of t-edges where one edge is labelled with  $\infty$  and the others with natural numbers. It replaces the  $\infty$  with the sum

of the two other edges. Both rules relabel an edge if there is a shorter path between the nodes of the edge. We have two *min* rules and two *sum* rules in order to distinct between the deletion/addition of  $\infty$  values and natural numbers. We need this distinction for the purpose of specifying the graph measures.

### *ShortDist*

initial: graph with edges between all nodes labelled with  $t : \infty$



control:  $(min_1|min_2)!; (sum_1|sum_2)!$

Figure 6.1: The Transformation Unit *ShortDist*

### Applying the Check-Algorithm

$$\begin{array}{l}
 check((min_1|min_2)!; (sum_1|sum_2)!) \\
 \Leftrightarrow check((min_1|min_2)!) \text{ and } check((sum_1|sum_2)!) \\
 \Leftrightarrow check((min_1|min_2)) \text{ and } N_{min_1|min_2} \text{ is not part. rep. and} \\
 \quad check(sum_1|sum_2) \text{ and } N_{sum_1|sum_2} \text{ is not part. rep.} \\
 \Leftrightarrow check(min_1) \text{ and } check(min_2) \text{ and } N_{min_1|min_2} \text{ is not part. rep. and} \\
 \quad check(sum_1) \text{ and } check(sum_2) \text{ and } N_{sum_1|sum_2} \text{ is not part. rep.} \\
 \Leftrightarrow T \text{ and } T \text{ and } N_{min_1|min_2} \text{ is not part. rep. and} \\
 \quad T \text{ and } T \text{ and } N_{sum_1|sum_2} \text{ is not part. rep.} \\
 \Leftrightarrow N_{min_1|min_2} \text{ is not part. rep. and } N_{sum_1|sum_2} \text{ is not part. rep.}
 \end{array}$$

In order to decide whether  $N_{min_1|min_2}$  respectively  $N_{sum_1|sum_2}$  is not partially repetitive we apply Lemma 83. Lemma 83 uses the incidence matrix of the

net, which is constructed by the change-vectors of the respective *alap* expression. As graph measures we employ the number of t-edges labelled with  $\infty$  (first component of change-vector), and the total amount of the values of all t-edges not labelled with  $\infty$  (second component of change-vector).

First we apply Lemma 83 to  $min_1|min_2$  and afterwards to  $sum_1|sum_2$ .

$$Change(min_1|min_2) = Change(min_1) \cup Change(min_2) = \{(-1, +a_1)\} \cup \{(0, -(z - a_2))\} = \{(-1, +a_1), (0, -(z - a_2))\}$$

$$A_{min_1|min_2} = \begin{pmatrix} -1 & 0 \\ +a_1 & -(z - a_2) \end{pmatrix}$$

According to Lemma 83 we have the following inequations:

$$\text{I } -1 * x_1 + 0 * x_2 \geq 0$$

$$\text{II } a_1 * x_1 - (z - a_2) * x_2 \geq 0$$

Dissolving the two inequations to  $x_1$  leads to:

$$\text{I } x_1 \leq 0$$

$$\text{II } x_1 \geq ((z - a_2) * x_2) \div a_1$$

Putting I and II together leads to  $((z - a_2) * x_2) / a_1 \leq x_1 \leq 0$

As we can read directly  $x_1 \leq 0$ . Since  $(z - a_2)$  and  $a_1$  greater than 0  $((z - a_2) * x_2) / a_1 \leq x_1 \leq 0$  implies that  $x_2 \leq 0$ . Since  $x_1$  and  $x_2$  must not be negative they both have to be 0. Thus  $N_{min_1|min_2}$  is not partially repetitive.

$$Change(sum_1|sum_2) = Change(sum_1) \cup Change(sum_2) = \{(0, -y) \mid y = z - (a_1 + b_1)\} \cup \{(-1, p) \mid p = a_2 + b_2\} = \{(0, -y), (-1, p) \mid y = z - (a_1 + b_1), p = a_2 + b_2\}$$

$$A_{sum_1|sum_2} = \begin{pmatrix} 0 & -1 \\ -y & p \end{pmatrix}$$

According to Lemma 83 we have the following two inequations:

$$\text{I } = 0 * x_1 - 1 * x_2 \geq 0 \text{ and}$$

$$\text{II } = -y * x_1 + p * x_2 \geq 0$$

Dissolve the two equations to  $x_2$  leads to

$$\text{I } = x_2 \geq (y * x_1) / p \text{ and}$$

$$\text{II } = x_2 \leq 0.$$

Putting these inequations together gives the following result:  $(y * x_1) / p \leq x_2 \leq 0$ . Since  $p > 0$  it follows  $x_1 \leq 0$  and  $x_2 \leq 0$ . Since  $x_1$  and  $x_2$  must not be negative they both have to be 0. This implies that  $N_{sum_1|sum_2}$  is not partially repetitive.



# Chapter 7

## Stepwise Controls

This chapter introduces a control condition, which allows to directly guide the graph transformation process. It is called *stepwise control*.

Control conditions cut down the non-determinism of rule application leading to a decreased amount of permitted derivations. Nevertheless this reduction most of the time does not apply to the time the computation of these permitted derivations takes. Often the semantics of a control condition is computed by building non-deterministically all possible derivations according to the given rules and then select those derivations which satisfy the control condition. Although this procedure cuts down the set of admitted derivations the time it takes to compute them may be still exponential. Various examples indicate that this approach is adequate on the level of modelling. But with respect to the explicit execution, it is not very helpful. From the viewpoint of execution, one would like to have a way of controlling the derivation process directly by the control condition so that derivations are only prolonged if they may end up to be permitted. If derivations take place in such a direct control mode, then the control condition determines which next steps are permitted at each current state of derivation.

A well known example of a direct control condition is a priority condition. Given a current graph, many rule applications may be possible, but only one is performed using a rule with highest priority. Another nice example of a direct control condition is a finite state automaton with rules as inputs. Beginning with the initial state of the automaton the following procedure is iterated starting with some initial graph: Choose a rule that has a state transition from the current state to a follow-up state which becomes the new current state, apply this rule to the current graph yielding the new current graph. The procedure can stop whenever a final state is reached. In this

case, the resulting derivation is permitted.

In the following we define, construct, and execute basic stepwise controls providing permitted derivations (respectively graph pairs) as semantics. In order to construct stepwise controls we introduce some basic compositions of stepwise controls. On the basis of these composition operators we transform already existing control conditions to stepwise controls. Particularly we introduce parallel stepwise controls which are able to model weak, synchronous, and proactive processes. The last section then equips transformation units with stepwise controls.

## 7.1 Basic Stepwise Controls

To get an image of a stepwise control one can imagine a finite state automaton extended by the ability to make a transition depending on some conditions. For example make a transition if a given rule is applicable to the current graph or only make a transition if a given rule is not applicable. Stepwise controls are additionally equipped with a set of regulation statements for steering their execution.

### 7.1.1 Definition and Construction

A stepwise control comprises a set of *states*, also called *control states*, including a *start state* and a set of *final states*. Control states are linked by transitions provided by a so called *guard relation*. The guard relation regulates the derivation process. Given a control state and a graph it provides a choice of *actions* or *regulation statements* leading to a result graph and a next control state. Actions may be any graph transforming devices, which provide a semantics consisting of graph pairs, in the simplest case they are rules. Regulation statements help to steer the derivation process and do not transform the input graph.

#### **Definition 84. Stepwise control**

A stepwise control  $SC$  is a tuple  $SC = (S, s_0, F, A, R, guard)$  with

- $S$  is a finite set of *control states*,
- $s_0 \in S$  is an *initial* control state,
- $F \subseteq S$  is a set of *final* control states,

- $A$  is a finite set of *actions* which provide a semantics  $SEM(x) \subseteq \mathcal{G} \times \mathcal{G} \forall x \in A$ ,
- $R$  is a set of *regulation statements*, and
- *guard* is a relation with  $guard \subseteq \mathcal{G} \times S \times (A \cup R) \times \mathcal{G} \times S$ . Mostly referred to as  $guard(G, s, x) \subseteq \mathcal{G} \times S$  or  $guard(G, s) \subseteq (A \cup R) \times \mathcal{G} \times S$  for  $G \in \mathcal{G}, s \in S$  and  $x \in A \cup R$ .

Given an input graph  $G$  and a control state  $s$ , the guard relation provides a selection of possible next transition steps potentially subject to some conditions regarding the input graph  $G$ . I.e. it provides in each case an action or regulation statement  $x$ , each leading to a graph  $G'$  and a next control state  $s'$ , where  $G'$  is the result graph of a graph transformation performed by the action  $x$  if  $x \in A$ , or  $G' = G$  if  $x$  is a regulation statement. If no transition step is possible the result of the guard relation is the empty set.

In the following we model some simple stepwise controls and afterwards introduce some basic composition operators for stepwise controls.

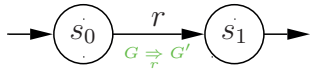
### Examples: Some simple stepwise controls

The following examples present some simple stepwise controls modelling a rule application, a rule application attempt, and an application of a rule as long as possible.

For the given examples let  $r \in \mathcal{R}$  be a rule and  $G, G' \in \mathcal{G}$  be graphs.

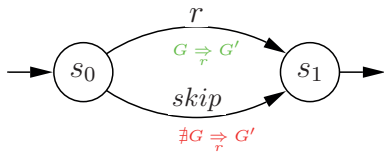
#### Rule application

$SC(r) = (\{s_0, s_1\}, s_0, \{s_1\}, \{r\}, \emptyset, guard)$  with  
 $guard(G, s_0, r) = \{(G', s_1) \mid G \xRightarrow[r]{G \Rightarrow G'}\}$ .



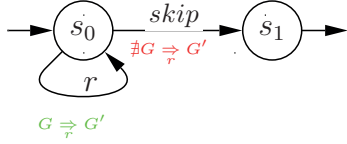
#### Try

$SC(try(r)) = (\{s_0, s_1\}, s_0, \{s_1\}, \{r, \{skip}\}, guard)$  with  
 $guard(G, s_0, r) = \{(G', s_1) \mid G \xRightarrow[r]{G \Rightarrow G'}\}$ , and  
 $guard(G, s_0, skip) = \{(G, s_1) \mid \neg \exists G \xRightarrow[r]{G \Rightarrow G'}\}$ .



### As-long-as-possible

$SC(r!) = (\{s_0, s_1\}, s_0, \{s_1\}, \{r\}, \{skip\}, guard)$  with  
 $guard(G, s_0, r) = \{(G', s_0) \mid G \Rightarrow_r G'\}$ , and  
 $guard(G, s_0, skip) = \{(G, s_1) \mid \nexists_r G \Rightarrow_r G'\}$ .



### Composition of stepwise controls

Like finite state automata, stepwise controls can be composed sequentially, in choice, and iteratively. The following definition implements these compositions adapting the respective definitions for automata.

#### Definition 85. Basic composition of stepwise controls

Let  $SC_i = (S_i, s_{0i}, F_i, A_i, R_i, guard_i)$  for  $i = 1, 2, 3$  be stepwise controls. Moreover, let  $G, G' \in \mathcal{G}$  be graphs.

#### Sequential composition

The sequential composition of  $SC_1$  and  $SC_2$  is defined by

$$SC_1; SC_2 = (S_1 \uplus S_2, s_{01}, F, A_1 \cup A_2, R_1 \uplus R_2, guard)$$

with

$$F = \begin{cases} F_1 \cup F_2 & \text{if } s_{02} \in F_2 \\ F_2 & \text{otherwise,} \end{cases}$$

$$guard(G, s, x) = guard_1(G, s, x), s \in S_1, x \in A_1 \cup R_1,$$

$$guard(G, s, x) = guard_2(G, s, x), s \in S_2, x \in A_2 \cup R_2, \text{ and}$$

$$guard(G, f, x) = \{(G', s') \mid f \in F_1, (G', s') \in guard_2(G, s_{02}, x), x \in A_2 \cup R_2\}.$$

#### Choice composition

The choice composition of  $SC_1$  and  $SC_2$  is defined by

$$SC_1 | SC_2 = (S_1 \uplus S_2 \uplus \{s_0\}, s_0, F, A_1 \cup A_2, R_1 \uplus R_2, guard)$$

with

$$F = \begin{cases} F_1 \cup F_2 \cup \{s_0\} & \text{if } s_{0i} \in F_i, i \in \{1, 2\} \\ F_1 \cup F_2 & \text{otherwise,} \end{cases}$$

$$guard(G, s_1, x_1) = guard_1(G, s_1, x_1) \forall s_1 \in S_1, x_1 \in A_1 \cup R_1,$$

$$guard(G, s_2, x_2) = guard_2(G, s_2, x_2) \forall s_2 \in S_2, x_2 \in A_2 \cup R_2,$$

$$guard(G, s_0, x) = \{(G', s') \mid (G', s') \in guard_1(G, s_{01}, x), x \in A_1 \cup R_1\}, \text{ and}$$

$$guard(G, s_0, x) = \{(G', s') \mid (G', s') \in guard_2(G, s_{02}, x), x \in A_2 \cup R_2\}.$$

## Iteration

The iteration of  $SC$  is defined by

$$\begin{aligned} SC_3^* &= (S_3 \cup \{s_{0*}\}, s_{0*}, F_3 \cup s_{0*}, A_3, R_3, guard_*) \text{ with} \\ s_{0*} &\notin S_3, \\ guard_*(G, s, x) &= guard_3(G, s, x) \\ guard_*(G, s_{0*}, x) &= guard_3(G, s_{03}, x), \text{ and} \\ guard_*(G, f, x) &= guard_3(G, s_{03}, x), f \in F_3. \end{aligned}$$

These composition operators can be employed to construct more complex stepwise controls from already existing ones. In Section 7.2.2 we employ them to recursively construct stepwise controls from given control conditions.

### 7.1.2 Execution of Stepwise Controls

An execution of a stepwise control begins at the start state with a given input graph. According to the guard relation possible steps are performed, each step resulting in a next control state and a new graph. When a final control state is reached the execution stops (for the current computation branch). The derivation induced by the computation is permitted by the stepwise control.

In order to make the execution tangible in a formal way we combine the current graph and the current control state to a *configuration*. Then the execution of a stepwise control can be considered as sequences of *computation steps* on configurations.

#### Definition 86. Configuration

Let  $SC = (S, s_0, F, A, R, guard)$  be a stepwise control.

A *configuration* is a pair  $(G, s)$  with  $G \in \mathcal{G}$  and  $s \in S$ . It is *initial* if  $s = s_0$  and *terminal* if  $s \in F$ .

#### Definition 87. Computation

Let  $SC = (S, s_0, F, A, R, guard)$  be a stepwise control.

- A *computation step* is given by  $(G, s) \vdash_x (G', s')$  if  $(G', s', x) \in choice(G, s)$  for some  $x \in A \cup R$ .  
(Also denoted by  $(G, s) \vdash (G', s')$  if  $x$  is not of interest.)

- A *computation* is a sequence of computation steps  $(G_0, s_0) \vdash_{x_1} (G_1, s_1) \vdash_{x_2} \dots \vdash_{x_n} (G_n, s_n)$ ,  $n \in \mathbb{N}$  starting at an initial configuration. (Abbreviated by  $(G_0, s_0) \vdash^{\mathbb{N}} (G_n, s_n)$  or  $(G_0, s_0) \vdash^* (G_n, s_n)$ .) The shortest computation is given by a configuration  $(G, s_0)$ .
- A computation  $(G_0, s_0) \vdash_{x_1} (G_1, s_1) \vdash_{x_2} \dots \vdash_{x_n} (G_n, s_n)$  is *complete* if  $(G_n, s_n)$  is a terminal configuration.

A computation is composed of computation steps, each performing an action modifying the current graph, or a regulation step, which does not alter the current graph. A computation contains all necessary information to provide a semantics of stepwise controls comprising permitted derivations or graph pairs.

### 7.1.3 Semantics of Stepwise Controls

Permitted graph pairs can straightforwardly be build from the set of complete computations. One only has to combine the input and result graphs of each computation to a pair.

**Definition 88. Semantics: Permitted graph pairs  $SEM_g$**

Let  $SC$  be a stepwise control condition. The set of all permitted graph pairs is defined by

$$SEM_g(SC) = \{(G, G') \mid (G, s_0) \vdash^* (G', s_n) \text{ is a complete computation}\}.$$

In order to provide permitted derivations, we have to transform the complete computations to derivations, i.e. we have to remove the regulation steps and the states and assemble the remaining information to a derivation. A derivation build from a computation is called *induced* derivation and defined as follows.

**Definition 89. Induced derivation**

Every computation of a stepwise control  $SC = (S, s_0, F, A, R, guard)$  induces a derivation recursively defined by

$$\overset{\Rightarrow}{d}((G, s) \vdash_x (G', s')) = \begin{cases} G \overset{\Rightarrow}{x} G' & \text{if } x \in A. \\ G \overset{\Rightarrow}{0} G & \text{otherwise.} \end{cases}$$

$$\overset{\Rightarrow}{d}((G, s) \vdash_x (G', s') \vdash^* (\bar{G}, \bar{s})) = \begin{cases} G \overset{\Rightarrow}{x} G' \circ \overset{\Rightarrow}{d}((G', s') \vdash^* (\bar{G}, \bar{s})) & \text{if } x \in A. \\ \overset{\Rightarrow}{d}((G', s') \vdash^* (\bar{G}, \bar{s})) & \text{otherwise.} \end{cases}$$

The semantics of a stepwise control providing a set of permitted derivations then is defined by the set of all induced derivations from complete computations.

**Definition 90. Semantics: Permitted derivations  $SEM_d$**

Let  $SC$  be a stepwise control condition. The set of all permitted derivations is defined by

$$SEM_d(SC) = \{\vec{d}((G_0, s_0) \vdash^* (G_n, s_n)) \mid (G_0, s_0) \vdash^* (G_n, s_n) \text{ is a complete computation}\}.$$

In order to obtain stepwise controls one can model them from scratch as it is done above for simple examples. Another option is to transform existing control conditions to stepwise controls, which is the focus of the next section.

## 7.2 Transform Given Control Conditions to Stepwise Controls

Given basic stepwise controls modelling rule applications, and the composition operators provided in the last section one can straightforwardly model regular expressions over rules as stepwise controls. In the following we construct stepwise controls for weak parallel and synchronous expressions, straightforwardly employing the automata constructions from Chapter 4. Also we construct stepwise controls from 'downgraded' *alap*-expressions, which are *alap*-expressions without the operators  $*$  and  $|$ . Finally, we introduce *parallel stepwise controls* executing weak parallel, synchronous, and proactive expressions.


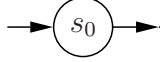
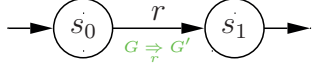
### 7.2.1 Weak and Synchronous Stepwise Controls

In Chapter 4 we have constructed finite state automata which execute weak and synchronous expressions. Since stepwise controls resemble finite state automata we could transfer these definitions to fit stepwise controls.

In order to model weak and synchronous expressions as stepwise controls we employ the composition operators introduced in Definition 85 and additionally define the weak and synchronous composition of stepwise controls.

**Definition 91. Weak and synchronous expressions to stepwise controls**

Let  $C, C_1, C_2$  be parallel expressions. Stepwise controls for parallel expressions are recursively defined by:

- $SC(\emptyset) = (\{s_0\}, s_0, \emptyset, \emptyset, \emptyset, \emptyset)$ ,  

- $SC(\lambda) = (\{s_0\}, s_0, \{s_0\}, \emptyset, \emptyset, \emptyset)$ ,  

- $SC(r) = (\{s_0, s_1\}, s_0, \{s_1\}, \{r\}, \emptyset, guard)$  with  
 $guard(G, s_0, r) = \{(G', s_1) \mid G \xrightarrow[r]{G'} G'\}$ ,  

- $SC(C_1; C_2) = SC(C_1); SC(C_2)$ ,
- $SC(C_1 \mid C_2) = SC(C_1) \mid SC(C_2)$ ,
- $SC(C^*) = SC(C)^*$ ,
- $SC(C_1 \$ C_2) = SC(C_1) \$ SC(C_2)$ ,
- $SC(C_1 \mathbb{W} C_2) = SC(C_1) \mathbb{W} SC(C_2)$ .

To define the weak parallel and synchronous composition of two stepwise controls we modify the definitions for the respective compositions of finite state automata (Definitions 53 and 54). Since stepwise controls for parallel expressions do not contain regulation statements, they do not have to be considered by the parallel composition, i.e. the set  $R$  stays empty.

**Definition 92. Weak parallel composition of stepwise controls**

Let  $SC_1 = (S_1, s_{01}, F_1, A_1, \emptyset, grd_1)$  and  $SC_2 = (S_2, s_{02}, F_2, A_2, \emptyset, grd_2)$  be two stepwise control conditions constructed from parallel expressions. The weak parallel composition of  $SC_1$  and  $SC_2$  is defined by

$SC_1 \mathbb{W} SC_2 = (S, s_0, F, A, \emptyset, grd)$  with

$$S = S_1 \times S_2,$$

$$s_0 = (s_{01}, s_{02}),$$

$$F = F_1 \times F_2,$$

$$A = (A_1 \mid \times \mid A_2) \cup A_1 \cup A_2,$$

$$R = \emptyset,$$

Simultaneous execution step with parallel rule composed of rules  $r_1$  from  $SC_1$



and  $r_2$  from  $SC_2$ :

$$\begin{aligned} \text{grad}(G, (s_1, s_2), r_1+r_2) = \{ & (G', (s'_1, s'_2)) \mid r_1 \in A_1, r_2 \in A_2, \\ & (G'_1, s'_1) \in \text{grad}_1(G, s_1, r_1), \\ & (G'_2, s'_2) \in \text{grad}_2(G, s_2, r_2), \\ & G \xRightarrow[r_1+r_2]{} G'\}, \end{aligned}$$

$SC_1$  operates alone:

$$\text{grad}(G, (s_1, s_2), r_1) = \{(G'_1, (s'_1, s_2)) \mid (G'_1, s'_1) \in \text{grad}_1(G, s_1, r_1), r_1 \in A_1\},$$

$SC_2$  operates alone:

$$\text{grad}(G, (s_1, s_2), r_2) = \{(G'_2, (s_1, s'_2)) \mid (G'_2, s'_2) \in \text{grad}_2(G, s_2, r_2), r_2 \in A_2\}.$$

The first equation of *guard* defines a simultaneous execution step with the parallel rule composed of the respective rules from the input stepwise controls. The second and the third equation define a transition step where only one of the input stepwise controls is operating.

### Definition 93. Synchronous composition

Let  $SC_1 = (S_1, s_{01}, F_1, A_1, \emptyset, \text{grad}_1)$  and  $SC_2 = (S_2, s_{02}, F_2, A_2, \emptyset, \text{grad}_2)$  be two stepwise control conditions constructed from parallel expressions. The synchronous composition of  $SC_1$  and  $SC_2$  is defined by

$SC_1 \$ SC_2 = (S, s_0, F, A, \emptyset, \text{grad})$  with

- $S = (S_1 \times S_2) \cup S_1 \cup S_2,$
- $s_0 = (s_{01}, s_{02}),$
- $F = F_1 \times F_2,$
- $A = (A_1 \mid \times \mid A_2) \cup A_1 \cup A_2,$
- $R = \emptyset,$  and
- transition step with a parallel rule:
$$\begin{aligned} \text{grad}(G, (s_1, s_2), r_1+r_2) = \{ & (G', (s'_1, s'_2)) \mid r_1 \in A_1, r_2 \in A_2, \\ & (G'_1, s'_1) \in \text{grad}_1(G, s_1, r_1), \\ & (G'_2, s'_2) \in \text{grad}_2(G, s_2, r_2), \\ & G \xRightarrow[r_1+r_2]{} G'\}, \end{aligned}$$

one input stepwise control operates further alone if the other has reached a final state:

$$\begin{aligned} \text{grad}(G, (s_1, s_2), x_1) &= \{(G', s'_1) \mid s_2 \in F_2, x_1 \in A_1, (G'_1, s'_1) \in \text{grad}_1(G, s_1, x_1)\}, \\ \text{grad}(G, (s_1, s_2), x_2) &= \{(G', s'_2) \mid s_1 \in F_1, x_2 \in A_2, (G'_2, s'_2) \in \text{grad}_2(G, s_2, x_2)\}, \end{aligned}$$

keep original guard relations:

$$\begin{aligned} \text{grad}(G, s_1, r_1) &= \{(G', s'_1) \mid (G'_1, s'_1) \in \text{grad}_1(G, s_1, r_1)\}, \\ \text{grad}(G, s_2, r_2) &= \{(G', s'_2) \mid (G'_1, s'_2) \in \text{grad}_2(G, s_2, r_2)\}. \end{aligned}$$

Please note, that the weak and synchronous composition of stepwise controls are only defined for stepwise controls build from weak and synchronous expressions.

The semantics of stepwise controls for weak parallel and synchronous expressions is defined as for basic stepwise controls.

Modelling proactive expressions in a similar way would result in an huge and complex stepwise control containing all possibilities to proactively compose the provided rules regarding the current graph, and the regulation necessary to organise the different cases. Therefore, in the next section we rather present a stepwise control implementing parallelism in a different way.

## 7.2.2 Parallel Stepwise Controls

A *parallel stepwise control* is a modified basic stepwise control able to model weak parallel, synchronous, and proactive processes. Different from stepwise controls for weak and synchronous expressions, it does not explicitly provide transitions labelled with parallel rules. The structure of a parallel stepwise control only provides the information which of its parts have to be executed in parallel, not how. The actual composition of the intended parallel rules takes place in the course of the stepwise control's computation.

In order to indicate which parts of the stepwise control have to be executed in parallel we employ hyperedges. Hyperedges do not only link two single states but sets of states, i.e. several states could be linked by only one edge. Considering parallel stepwise controls a hyperedge from a single state to a set of states indicates that all the target states have to be considered simultaneously during execution of the stepwise control. Vice versa a hyperedge from a set of states to a single state indicates that the parallel consideration of the source states ends.

A parallel stepwise control does not differ much from a basic stepwise control. One difference is that, due to the hyperedges, its guard relation operates on sets of states instead of single states. The other difference is that a parallel stepwise control contains a set of waiting states  $W$ .  $W$  comprises states in which the stepwise control explicitly is allowed to wait, i.e. it does not have to perform a rule application while executed in parallel with others.

We need explicit waiting states due to synchronous composition of parallel stepwise controls. This is explained in more detail when introducing parallel composition (Definition 96).

**Definition 94. Parallel stepwise control**

A *parallel stepwise control* is a tuple  $p\mathcal{SC} = (S, s_0, F, W, A, R, guard)$  where

- $S, s_0, F, A, R$ , are defined as for basic stepwise controls (Def. 84),
- $W \subseteq S$  is a set of waiting states, and
- $guard$  is the transition relation operating on sets of states,  
 $guard \subseteq (\mathcal{G} \times \mathcal{P}(S) \times (A \cup R) \times \mathcal{G} \times \mathcal{P}(S))$ .

The following definition recursively constructs a parallel stepwise control from a given parallel expression.

**Definition 95. Parallel stepwise control from parallel expression**

Let  $C$  be a proactive, weak, or synchronous expression. The parallel stepwise control implementing  $C$ ,  $p\mathcal{SC}(C)$ , is recursively defined by:

- $p\mathcal{SC}(\emptyset) = (\{s_0\}, s_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ ,
- $p\mathcal{SC}(\lambda) = (\{s_0\}, s_0, \{s_0\}, \emptyset, \emptyset, \emptyset, \emptyset)$ ,
- $p\mathcal{SC}(r) = (\{s_0, s_1\}, s_0, \{s_1\}, \emptyset, \{r\}, \emptyset, guard)$   
 $guard(G, \{s_0\}, r) = \{(G', \{s_1\}) \mid G \xRightarrow[r]{\phantom{r}} G'\}$ ,
- $p\mathcal{SC}(C_1; C_2) = p\mathcal{SC}(C_1); p\mathcal{SC}(C_2)$ ,
- $p\mathcal{SC}(C_1 \mid C_2) = p\mathcal{SC}(C_1) \mid p\mathcal{SC}(C_2)$ ,
- $p\mathcal{SC}(C^*) = p\mathcal{SC}(C)^*$ ,
- $p\mathcal{SC}(C_1 \# C_2) = p\mathcal{SC}(C_1) \# p\mathcal{SC}(C_2)$ ,
- $p\mathcal{SC}(C_1 \mathbb{W} C_2) = p\mathcal{SC}(C_1) \mathbb{W} p\mathcal{SC}(C_2)$ ,
- $p\mathcal{SC}(C_1 \$ C_2) = p\mathcal{SC}(C_1) \mathbb{W} p\mathcal{SC}(C_2)$ .

The sequential, choice, and iterative composition are defined analogously to basic stepwise controls (Definition 85), apart from employing sets of states instead of single states in the guard relation and taking into account the waiting states by uniting the waiting states of the input stepwise controls. Since the differences are marginal we do not provide the definitions explicitly. The weak, synchronous, and proactive composition are all mapped to

the parallel composition of stepwise controls,  $p\mathcal{SC}(C_1) \parallel p\mathcal{SC}(C_2)$ , implementing the general parallel structure.

In order to compose two stepwise controls in parallel we add a new start state and link it to the former start states with a *begin*|-hyperedge, indicating their parallel association. To enable an input stepwise control to wait, when its execution has come to an end we add copies of all former final states, serving as waiting states. Every former final state then is linked to its copy by an transition labelled with *wait*. In order to quit the parallel execution of the two input stepwise controls we link by pairs of two the waiting states (one originated in  $F_1$  and one in  $F_2$ ) by a hyperedge labelled with *end*|| to a new final state. The new set of waiting states is comprised of the former waiting states and the new ones. We do not use the former final states themselves as waiting states for two reasons. First of all, the explicit distinction of final states from the other states is lost when composing the input stepwise controls. Hence, we need another way to distinct waiting states from the other states, e.g., another explicitly given set. If we would use the former final states collected in such a set it could be the case that a final state also has another outgoing transition, e.g., labelled with a rule. Allowed to wait a stepwise control could pause execution if the respective rule is not applicable and continue if it is. This is forbidden when composing two stepwise controls synchronously. Consider the same situation employing copies of former final states as waiting states. Now, being in a former final state the stepwise control can choose to continue execution choosing the transition with the rule, or stop execution by making the transition to the respective waiting state. (Since the technical realisation of a copy operation would make the following definition confusing we do not explicitly define and use such a function. Instead we use a "dummy"-function  $copy(state)$ , considered to provide a copy of  $state$  which could be recognised as a copy of  $state$ .)

**Definition 96.**  $p\mathcal{SC} \parallel p\mathcal{SC}$

Let  $p\mathcal{SC}_1 = (S_1, s_{1_0}, F_1, W_1, A_1, R_1, grd_1)$  and  $p\mathcal{SC}_2 = (S_2, s_{2_0}, F_2, W_2, A_2, R_2, grd_2)$  be two parallel stepwise controls. The parallel composition of  $p\mathcal{SC}_1$  and  $p\mathcal{SC}_2$  is defined by:

$p\mathcal{SC}_1 \parallel p\mathcal{SC}_2 = (S, s_0, \{s_f\}, W, A, R, grd)$  with

- $S = S_1 \uplus S_2 \uplus \{s_0, s_f\} \cup copy(F_1) \cup copy(F_2)$ , Since  $F_1$  and  $F_2$  are subsets of  $S_1$  respectively  $S_2$  we assume  $copy(F_1)$  and  $copy(F_2)$  to be disjoint.
- $W = W_1 \cup W_2 \cup copy(F_1) \cup copy(F_2)$ , Since  $W_1$  and  $W_2$  are subsets of  $S_1$  respectively  $S_2$  we assume  $W_1$  and  $W_2$  to be disjoint and, as above,

$copy(F_1)$  and  $copy(F_2)$  to be disjoint.

- $A = A_1 \cup A_2$ ,
- $R = R_1 \cup R_2 \cup \{begin||, end||, wait\}$ ,
- Hyperedge from new start state to both old start states to indicate their parallel connection:  
 $grad(G, \{s_0\}, begin||) = \{(G, \{s_{1_0}, s_{2_0}\})\}$ ,

Hyperedges from each possible pair of waiting states (one associated with  $p\mathcal{C}_1$  the other with  $p\mathcal{C}_2$ ) to new final state in order to quit parallel connection:

$$grad(G, \{s_1, s_2\}, end||) = \{(G, \{s_f\})\}, s_1 \in copy(F_1), s_2 \in copy(F_2),$$

Keep original guard relations:

$$grad(G, \{s\}, x) = \{(G', \{s'\}) \mid (G', \{s'\}) \in grad_1(G, \{s\}, x) \cup grad_2(G, \{s\}, x)\},$$

Keep original hyperedges:

$$\begin{aligned} grad(G, \{s\}, begin||) &= \{(G, \{s_1, s_2\}) \mid (G, \{s_1, s_2\}) \in \\ &\quad grad_1(G, \{s\}, begin||) \cup grad_2(G, \{s\}, begin||)\}, \\ grad(G, \{s_1, s_2\}, end||) &= \{(G, \{s\}) \mid (G, \{s\}) \in \\ &\quad grad_1(G, \{s_1, s_2\}, end||) \cup grad_2(G, \{s_1, s_2\}, end||)\}, \end{aligned}$$

Link all old final states to their respective waiting state:

$$grad(G, f, wait) = \{(G, f') \mid f' = copy(f)\} \forall f \in F_1 \cup F_2.$$

For illustration, Figure 7.1 depicts a schematic representation of the parallel composition of two stepwise controls. The "assembly" of hyperedges is denoted by a dot, in order to distinguish the assembly from an arbitrary intersection of edges. The input stepwise controls are depicted in grey.

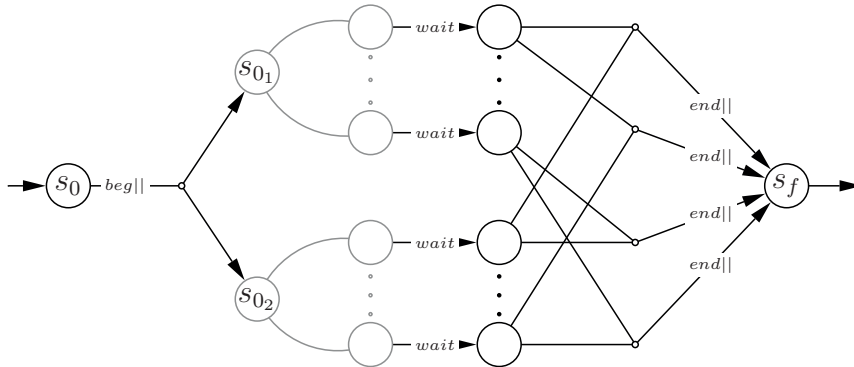


Figure 7.1: Parallel composition of two stepwise controls

### Parallel computation

Parallel computation realises the execution of a parallel stepwise control. It operates on sets of states, all of which have to be considered in each computation step. According to the kind of computation step some of these states make a transition (e.g., the computation performs a regulation step for one of the states, or it performs a synchronous transition step, where all states have to make a transition.) The next current state is given by the follower states of the performed transitions and the states that made no transition. The next current graph is obtained by applying the respective (parallel) rule if the transitions were labelled with rules, or the current graph stays the same if the computation has performed a regulation step.

Formally, a parallel computation runs on *parallel configurations* comprising the current graph and the set of current states.

#### Definition 97. Parallel configuration

Let  $p\mathcal{SC} = (S, s_0, F, W, A, R, guard)$  be a parallel stepwise control.

A *parallel configuration* is a pair  $(G, cS)$  with  $G \in \mathcal{G}$  is the current graph and  $cS \subseteq S$  is the set of current states.

It is *initial* if  $cS = \{s_0\}$  and *terminal* if  $cS = \{s_f\}$  with  $s_f \in F$ .

The parallel computation performs a single rule application or regulation step analogously to basic stepwise controls: considering a current state a transition is chosen and performed. In order to prevent the application of a single rule when parallelism is demanded, a single rule application only can be performed if the set of current states contains only this single rule. To keep the reading simple in the following we call "the set of current states"  $cS$ .

In order to perform a parallel rule application a subset  $S_{act} \subseteq cS$  is chosen according to the parallel form (weak, proactive, or synchronous). This set contains all active states that make a transition in the current computation step. The passive states which make no transition are assembled in a set called  $S_{pass}$ . For weak parallel composition  $S_{act}$  is an arbitrary subset of  $cS$ . Considering proactive composition it is a subset of  $cS$  subject to the following condition: After choosing a rule-labelled transition for each state in  $cS$ , each parallel rule composed of the respective rules and one additional rule provided by a transition for some state in  $S_{pass}$  is no more applicable to the current graph. For synchronous composition  $S_{act}$  contains all states of  $cS$ , that are no waiting states.

For each state from  $S_{act}$  one transition is chosen providing a rule and a next state. (Please note, that there may be several possibilities to make such a choice, since the considered stepwise controls are non-deterministic.) The obtained rules are composed to a parallel rule and applied to the current graph, leading to the result graph of the computation step. All the next states provided by the chosen transitions and the states from  $S_{pass}$  constitute the next current state.

Since a parallel stepwise control is recursively constructed the information, which of its parts have to be considered in parallel, is encoded in the structure of the stepwise control by successive *begin*||-hyperedges. Hence, the computation may only perform a parallel rule application if all states, associated by successive *begin*||-hyperedges, have been collected. In order to collect all these states beforehand the computation has to prefer transitions with *begin*||-hyperedges. This vice versa applies analogously to *end*||-hyperedges.

**Definition 98. Parallel computation steps**

Let  $pSC = (S, s_0, F, W, A, R, guard)$  be a parallel stepwise control. The parallel computation steps on parallel configurations are defined as follows.

Regulation step

$$(G, cS) \vdash_x (G, cS')$$

for some  $(G, \{s'\}) \in grd(G, \{s\}, x)$ ,  $s \in cS$ ,  $x \in R$  and  
 $cS' = (cS \setminus \{s\}) \cup \{s'\}$ .

Single (non parallel) rule application

$$(G, \{s\}) \vdash_x (G', \{s'\})$$

for some  $(G', \{s'\}) \in grd(G, \{s\}, x)$ ,  $x \in A$ .

### Begin parallel execution

$$(G, cS) \vdash (G, cS') \quad \text{begin||}$$

if  $(G, \{s_1, s_2\}) \in \text{grad}(G, \{s\}, \text{begin||})$ , for some  $s \in cS$  and  $cS' = (cS \setminus \{s\}) \cup \{s_1, s_2\}$ .

### Quit parallel execution

$$(G, cS) \vdash (G, cS') \quad \text{end||}$$

if  $(G, \{s\}) \in \text{grad}(G, \{s_1, s_2\}, \text{end||})$ , for some  $s_1, s_2 \in cS$  and  $cS' = (cS \setminus \{s_1, s_2\}) \cup \{s\}$ .

### Proactive rule application

$$(G, cS) \vdash (G', cS')$$

Choose set of active states and a transition for each active state

Choose  $S_{act} \subseteq cS$  and for all  $s \in S_{act}$  a transition

$$(G', \text{next}_s, r_s) \in \text{grad}(G, s) \text{ with } r_s \in A.$$

The parallel rule composed of rules provided by the chosen transitions is applicable to  $G$ , resulting in  $G'$

$$\text{Then } G \xRightarrow{\sum_{s \in S_{act}} r_s} G' \text{ and}$$

No parallel rule application with an additional rule possible

$$\nexists G \xRightarrow{\sum_{s \in S_{act}} (r_s) \vdash r} G'' \text{ for a rule } r \text{ provided by some transition}$$

$$(\bar{G}, \bar{s}, r) \in \text{grad}(G, s_{pass}), s_{pass} \in cS \setminus S_{act} \text{ and}$$

No *begin||* or *end||* transition possible for passive states

$$\nexists (G, \{s_1, s_2\}) \in \text{grad}(G, \{s\}, \text{begin||}) \quad \forall s \in (cS \setminus S_{act}) \text{ and}$$

$$\nexists (G, \{s\}) \in \text{grad}(G, \{s_1, s_2\}, \text{end||}) \quad \forall s_1, s_2 \in (cS \setminus S_{act}) \text{ and}$$

The next parallel state is given by the next states of the chosen transitions and the states that made no transition

$$cS' = (cS \setminus S_{act}) \cup \{\text{next}_s \mid s \in S_{act}\}.$$

### Weak parallel rule application

$$(G, cS) \vdash (G', cS')$$

Choose  $S_{act} \subseteq cS$  and for all  $s \in S_{act}$  a transition

$$(G', \text{next}_s, r_s) \in \text{grad}(G, s), r_s \in A.$$

$$\text{Then } G \xRightarrow{\sum_{s \in S_{act}} r_s} G' \text{ and}$$

No *begin||* or *end||* transition possible for passive states

$$\nexists (G, \{s_1, s_2\}) \in \text{grad}(G, \{s\}, \text{begin||}) \quad \forall s \in (cS \setminus S_{act}) \text{ and}$$

$$\nexists (G, \{s\}) \in \text{grad}(G, \{s_1, s_2\}, \text{end||}) \quad \forall s_1, s_2 \in (cS \setminus S_{act}) \text{ and}$$



$$cS' = (S \setminus S_{act}) \cup \{next_s \mid s \in S_{act}\}.$$

### Synchronous rule application

$$(G, cS) \vdash (G', cS')$$

Choose a transition for all states that are no waiting states

Let  $S_{act} = cS \setminus W$ . Choose for all  $s \in S_{act}$  a transition

$$(G', next_s, r_s) \in grd(G, s), r_s \in A.$$

Then  $G \xRightarrow{\sum_{s \in S_{act}} r_s} G'$  and

No  $begin||$  or  $end||$  transition possible for passive/waiting states

$$\nexists (G, \{s_1, s_2\}) \in grd(G, \{s\}, begin||) \forall s \in W \text{ and}$$

$$\nexists (G, \{s\}) \in grd(G, \{s_1, s_2\}, end||) \forall s_1, s_2 \in W \text{ and}$$

$$cS' = W \cup \{next_s \mid s \in S_{act}\}.$$

### Induced derivation

The induced derivation of a parallel computation is analogously defined to basic stepwise controls in Definition 89. The semantics of a parallel stepwise control is defined as for basic stepwise controls in Definition 88 and 90.

#### Definition 99. Induced derivation for parallel computation

Every parallel computation of a parallel stepwise control

$pSC = (S, s_0, F, W, A, R, guard)$  induces a derivation recursively defined by

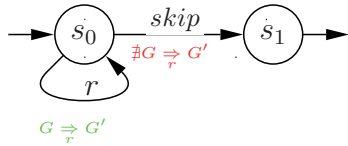
$$\xRightarrow{d((G, cS) \vdash_x (G', cS'))} = \begin{cases} G \xRightarrow{x} G' & \text{if } x \in A_*. \\ G \xRightarrow{0} G & \text{otherwise.} \end{cases}$$

$$\xRightarrow{d((G, cS) \vdash_x (G', cS'))^* (\bar{G}, \bar{cS})} = \begin{cases} G \xRightarrow{x} G' \circ \xRightarrow{d((G', cS') \vdash^* (\bar{G}, \bar{cS}))} & \text{if } x \in A_*. \\ \xRightarrow{d((G', cS') \vdash^* (\bar{G}, \bar{cS}))} & \text{otherwise.} \end{cases}$$

### 7.2.3 As-long-as-possible Stepwise Controls

Another control condition we want to model as stepwise control is as-long-as-possible. In the following we try to construct a stepwise control from an *alap*-expression as introduced in Chapter 5. Thereby we encounter some problems and avoid them by omitting the operators  $*$  and  $|$ . The resulting expression, called *alap*<sup>-</sup>-expression (reduced *alap*-expression), then is modelled as stepwise control.

In 7.1.1 we have constructed a stepwise control from scratch executing a rule  $r$  as long as possible. The stepwise control iterates the application of  $r$  as long as  $r$  is applicable to the current graph. In case  $r$  is no more applicable it makes a transition to its final state.



In order to transfer this idea to entire *alap*-expressions we consider the following idea: Model the expression, which has to be iterated as long as possible, as stepwise control. Extend the resulting stepwise control in such a way that it starts a new iteration whenever it has come to an end; and in case it gets stuck (without being in a final state), it aborts the iteration process if there is no other (unexplored) possibility to accomplish a complete run.

Unfortunately, in case the expression contains  $*$  there may be infinitely many other potential possibilities to complete a run created by  $*$ . Hence, the question whether there is no other possibility to accomplish a run could sometimes never be answered positively, since infinitely many possibilities would have to be considered. Therefore, we omit the  $*$ -operator. In case the potential runs are only created by (finite many)  $|$ -operators it is possible to decide if all options have been computed and failed. Although theoretically possible, we also omit  $|$  since the organisation of the computation would be too complex. The reintegration of the operator  $|$  is left for future work.

The resulting expression is called *alap<sup>-</sup>-expression*. Iterating an *alap<sup>-</sup>*-expression as long as possible, one can immediately abort the iteration process whenever a rule application fails, since there could not be further options which would have to be explored first. Then the new current graph has to be the graph present at the begin of the failed run. In order to 'track back' to this graph during computation, the computation has to memorise the current graph each time a new run begins. In order to enable the computation to do this the *alap<sup>-</sup>*-stepwise control has to indicate the begin of a new run.

The above informally described approach constitutes the as-long-as-possible composition step in the recursive construction of a stepwise control from an *alap<sup>-</sup>*-expression. The as-long-as-possible composition extends the given stepwise control by the ability to initiate new runs of itself and abort the iteration if it gets stuck. It comprises a new start and a new final state. The new start state is linked to the former start state by a new transition labelled with *alap*, indicating the begin of an iterated execution as long as possible. (Encountering such a transition the computation memorises the current graph.)

Every state of the origin stepwise control (which is not a dead end) is linked with a transition, labelled with *abort*, to the the new final state. Such an *abort*-transition only is enabled if none of the other transitions is pursuable. In order to initiate a new run the former final state is linked with the former start state by a new transition labelled with *newrun* (like for the transition *alap*, encountering *newrun* the computation memorises the current graph). Formally the as-long-as-possible composition of an *alap*<sup>-</sup>-stepwise control is defined as follows.

**Definition 100. *SC*! as-long-as-possible composition**

Let  $SC = (S, s_0, F, A, R, grd)$  be a stepwise control. The as-long-as-possible composition,  $SC!$ , is defined by:

$$SC! = (S \uplus \{s_0^!, s_f^!\}, s_0^!, \{s_f^!\}, A, R \cup \{alap, abort, newrun\}, grd^!) \text{ with}$$

initiate *alap*

$$grd^!(G, s_0^!, alap) = \{(G, s_0)\} \forall G \in \mathcal{G},$$

maintain *guard* relation from origin *SC*

$$grd^!(G, s, x) = \{(G', s') \mid (G', s') \in grd(G, s, x)\},$$

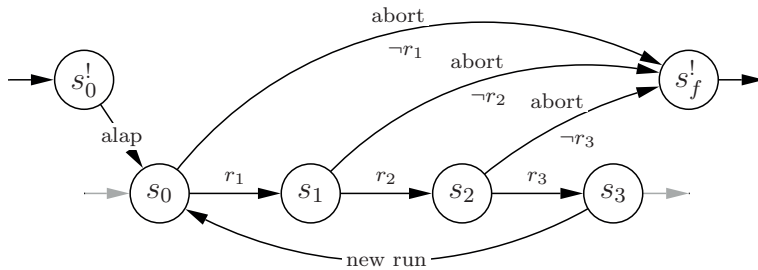
when a final state of the origin *SC* is reached initiate a new iteration

$$grd^!(G, s, newrun) = \{(G, s_0)\} \forall s \in F, G \in \mathcal{G},$$

if a rule provided by guard is not applicable to current graph, abort to final state

$$grd^!(G, s, abort) = \{(G, s_f) \mid \exists(\bar{G}', s') \in grd(\bar{G}, s, x), x \in (A \cup R), s \in (S \setminus F) \text{ but } \nexists(G', s') \in grd(G, s, x)\}.$$

The following illustration depicts an *alap*-stepwise control. Due to space limitations we have abbreviated the statement "if  $\nexists G \Rightarrow G'$ " by  $\neg r_1$ .



Please note, that this construction would not work having final states with outgoing transitions. Fortunately, due to construction of  $alap^-$ -stepwise controls there are no final states with outgoing transitions.

### Computation of $alap^-$ -expressions

As explained above the computation has to continue the computation with the proper graph in case the current run fails. Therefore it has to memorise the current graph before each new run. This is done within the configuration, which definition is adapted respectively.

#### Definition 101. Configuration of $alap^-$ -stepwise control

Let  $SC = (S, s_0, F, A, R, grd)$  be an  $alap^-$ -stepwise control.

A configuration of  $SC$  is a tuple  $(G, s, \sigma)$  with  $G \in \mathcal{G}$  is the current graph,  $s \in S$  is the current state, and  $\sigma$  is a sequence of memorised graphs over  $\mathcal{G}$ .

In case there are no memorised graphs  $\sigma$  is empty, denoted by  $\lambda$ . Otherwise the most left graph of  $\sigma$  is the latest memorised graph.

Initial and terminal configurations are defined analogously to basic stepwise controls as well as computation and complete computation.

The computation steps realise the execution of the  $alap^-$ -stepwise controls. Encountering an  $alap$ -labelled transition, which indicates a succeeding part of the stepwise control to be iterated as long as possible, the computation memorises the current graph, i.e. the current graph is appended to the left of the sequence of memorised graphs. A rule application is performed analogously to basic stepwise controls, whereby the memorised graphs remain unaffected. Encountering a *newrun*-transition the latest memorised graph is replaced by the current graph. Meeting *abort* the computation continues with the latest memorised graph, i.e. the leftmost graph is taken from the sequence and serves as new current graph.

#### Definition 102. Computation steps

Let  $G, G' \in \mathcal{G}$  be graphs and  $SC = (S, s_0, F, A, R, grd)$  an  $alap^-$ -stepwise control. Computation steps on configurations realising the execution of  $alap^-$ -stepwise controls are defined as follows.

- Memorise current graph if swc initiates new  $alap$   
 $(G, s, \sigma) \vdash_{alap} (G, s', G\sigma)$  if  $(G, s', alap) \in grd(G, s)$ .
- Rule application step  
 $(G, s, \sigma) \vdash_x (G', s', \sigma)$  if  $(G', s', x) \in grd(G, s), x \in A$ .

- Memorise current graph when swc initiates new run  
 $(G, s, G'\sigma) \vdash_{newrun} (G, s', G\sigma)$  if  $(G, s', newrun) \in grd(G, s)$ .
- Continue with memorised graph if swc aborts current run  
 $(G, s, G'\sigma) \vdash_{abort} (G', s', \sigma)$  if  $(G, s', abort) \in grd(G, s)$ .

### Induced derivation

In order to construct the induced derivation from an  $alap^-$ -computation, we process the computation from the end. If we would construct the induced derivation starting from the beginning of the computation, like it is done for basis stepwise controls, we would have to discard the last piece of the so far constructed derivation, when encountering an *abort*. Encountering an abort means that the last steps of the computation are not used for the permitted derivation since the last iteration failed.

The problem is that the construction process for the induced derivation works recursively, with each recursion step having the scope of a direct derivation. When encountering an *abort*, the part of the derivation to which the computation would have to track back to is no more accessible.

Starting with the result graph and constructing the derivation from the end we are able to construct only derivation parts actually contained in the permitted derivation. Encountering an *abort* we can continue the construction process at that point of the computation where the failed run begins and skip the computation inbetween. The following definition formalises this proceeding.

#### Definition 103. Induced derivation of alap Computation

Let  $SC = (S, s_0, F, A, R, grd)$  be an  $alap^-$ -stepwise control. Moreover, let  $comp$  be a computation of  $SC$ . The induced derivation of  $comp$ ,  $\overset{\Rightarrow}{d}(comp)$ , is recursively defined by

$$\begin{aligned}
\text{(i)} \quad & \overset{\Rightarrow}{d}((G, s, \lambda)) = G \overset{0}{\Rightarrow} G, \\
\text{(ii)} \quad & \overset{\Rightarrow}{d}(\dots(G_{n-1}, s_{n-1}, \sigma) \vdash_{x_n} (G_n, s_n, \sigma')) = \begin{cases} \overset{\Rightarrow}{d}(\dots(G_{n-1}, s_{n-1}, \sigma)) \circ G_{n-1} \overset{x_n}{\Rightarrow} G_n & \text{if } x_n \in A, \\ \overset{\Rightarrow}{d}(\dots(G_{n-1}, s_{n-1}, \sigma)) & \text{if } x_n \in R \setminus \{abort\}, \end{cases}
\end{aligned}$$

- (iii) Encountering *abort*, skip the computation up to the latest *newrun* resp. *alap*
- $$\begin{aligned}
& \xRightarrow{alap} \\
\text{a) } & d(\dots(G_i, s_i, G'\sigma) \vdash_{new} \dots(G_j, s_j, G\sigma) \vdash_x (G_{j+1}, s_{j+1}, G\sigma) \dots \vdash_{abort}(G_n, s_n, \sigma)) \\
& = d(\dots(G_i, s_i, G'\sigma)) \quad \text{if } x \notin \{alap, newrun\} \forall j \in \mathbb{N}, i < j < n, \\
& \xRightarrow{} \\
\text{b) } & d(\dots(G_i, s_i, \sigma) \vdash_{alap} \dots(G_j, s_j, G\sigma) \vdash_x (G_{j+1}, s_{j+1}, G\sigma) \dots \vdash_{abort}(G_n, s_n, \sigma)) \\
& = d(\dots(G_i, s_i, \sigma)) \quad \text{if } x \notin \{alap, newrun\} \forall j \in \mathbb{N}, i < j < n.
\end{aligned}$$

## 7.3 Transformation Units and Stepwise Controls

The following section examines how to relate stepwise controls and transformation units. The first part employs a stepwise control as control condition for a transformation unit. The second part models a transformation unit itself as stepwise control.

### 7.3.1 Stepwise Controls for Transformation Units

In the preceding examples we already used stepwise controls as control conditions for simple transformation units, as this is done straightforwardly. The set of actions of the stepwise control corresponds to the set of rules of the unit and the guard relation implements the respective rule application instructions.

In order to employ stepwise controls as control conditions for structured transformation units we have to determine how the computation of an imported unit is embedded into the computation of the calling unit. A simple approach is to consider each execution of an imported unit as single computation step of the calling unit and do not mind its individual computation. The imported units then are actions of the main unit and the guard relation implements the call of an imported unit by a transition labelled with this unit, employing the graph-pair-semantics of the imported unit.

In case all the imported units themselves are equipped with stepwise controls one can also execute the imported units in a stepwise manner and embed their computation explicitly in the computation of the calling unit. Now, whenever an imported unit is called, one may interrupt the running process and start a stepwise computation in the imported unit. If this is terminated, one

can return to the interrupted computation and continue it with the result of the import process. Altogether, one gets a computational process which control component is organised as a stack of configurations, each configuration representing a unit and its current execution state. The computational steps are rule applications and stack operations. To start a computation of an imported unit, a configuration containing the unit itself, its initial control state, and the current graph is pushed on top of the control stack. To stop a computation of an imported unit, one can return to the configuration of the underlying computation by application of the pop operation. In the following this idea is formalised.

A *stack configuration* holds all required information regarding respectively one of the involved transformation units: the unit itself, the current state of its stepwise control and the current graph.

**Definition 104. Stack configuration**

A *stack configuration* is a tuple  $(G, tu, s)$  with  $G \in \mathcal{G}$ ,  $tu$  is a transformation unit with a stepwise control  $C$ , and  $s \in S_C$ . Initial and terminal configuration are defined analogously to Definition 86.

The configurations are organised in a *stack of configurations*. The configuration of the currently executed unit is on top of the stack, its calling unit one configuration deeper and so on.

**Definition 105. Stack of configurations**

A *stack of configurations* is a sequence of stack configurations  $(G_1, tu_1, s_1) \dots (G_n, tu_n, s_n)$  with  $G_i \in \mathcal{G}$ ,  $tu_i$  are transformation units with stepwise controls  $C_i$ , and  $s_i \in S_{C_i}$  for  $i \in \mathbb{N} \setminus \{0\}$ .

The *computation steps* realise the organisation of the execution of the (imported) units. In every computation step the configuration on the top, i.e. the current unit, is processed. According to the actual situation, one of the provided computation steps is performed. Whenever an imported unit is called its configuration is pushed onto the stack and the computation continues executing this unit. When its execution has terminated, i.e. its control condition is satisfied and it has reached a terminal graph, the configuration representing the imported unit is pulled from the stack and the computation continues with the underlying unit. Rule applications or regulation statements are performed analogously to the respective computation steps for basic stepwise controls, addressed in Definition 87.

**Definition 106. Computation steps on stack of configurations**

Let  $tu_1, tu_2, tu''$  be transformation units with stepwise controls  $C_1$  and  $C_2, C''$ . Moreover, let  $G, G''$ ,  $\overset{egal}{G} \in \mathcal{G}$  be graphs.

- (i) Rule application or regulation step  
 $(G, tu_1, s)(G'', tu_2, s'') \cdots \vdash (G', tu_1, s')(G'', tu_2, s'') \dots$  for  
 $(G', s') \in guard_{C_1}(G, s, a), a \in A_{C_1} \cup R_{C_2}$ .  
 The stepwise control of the current unit  $tu_1$  makes a transition with an action or a regulation statement.
- (ii) New imported unit  
 $(G, tu_1, s) \cdots \vdash (G, tu_2, s_{0C_2})(G, tu_1, s') \dots$  if  
 $(G, s') \in guard_{C_1}(G, s, tu_2), G \in SEM(I_{tu_2})$   
 A transition, labelled with an imported unit  $tu_2$ , is reached by the stepwise control of the current unit  $tu_1$ . If the current graph satisfies the initial graph class expression of  $tu_2$ , the stepwise control of  $tu_1$  makes a step to its following state, and a new configuration representing the imported unit  $tu_2$  is pushed onto the stack.
- (iii) Imported unit complete  
 $(G, tu_2, f)(G', tu_1, s') \cdots \vdash (G, tu_1, s') \dots$  if  
 $f \in F_{C_2}$  and  $G \in SEM(T_{tu_2})$ .  
 The stepwise control of the current unit has reached a final state. Then, provided the current graph is permitted by the terminal graph class expression of the current unit, its configuration is pulled from the stack and the computation continues with the underlying configuration and the current graph.

### 7.3.2 Transformation Unit as Stepwise Control

The last section has shown how to employ a stepwise control as control condition for a transformation unit. Now, this section introduces how to model an entire transformation unit itself as stepwise control.

In order to employ a transformation unit as stepwise control, the control condition of the unit has to be a stepwise control. Moreover, it has to be ensured that the initial and terminal graph class expressions of the unit are satisfied. To achieve this we add a new initial state to the units stepwise control, linked by a new transition to the former initial state, only allowing a transition step provided that the input graph satisfies the initial graph class expression of the unit. Analogously we add a new final state linked by transitions from the former final states, ensuring that the result graph satisfies the terminal graph class expression. The following definition presents the construction of a stepwise control from a transformation unit. It applies to simple and structured transformation units, since on control condition level



the imported units are handled like rules. I.e. the rules and imported units are all actions of the unit's stepwise control condition and handled equally.

**Definition 107. Transformation unit to stepwise control**

Let  $tu = (I, P, C, T) \in \mathcal{TU}$  or  $tu = (I, P, U, C, T) \in \mathcal{TU}$  be a transformation unit with a stepwise control condition  $C = (S_C, s_{0C}, A_C, R_C, F_C, guard_C)$ .

The stepwise control based on  $tu$  is defined by:

$SC(tu) = (S, s_0, F, A, guard)$  with

$S = S_C \cup \{s_0, f\}$  with  $s_0, f \notin S_C$ ,

$F = \{f\}$ ,

$A = A_C \cup \{skip\}$ ,

$guard(G, s_0, skip) = \{(G, s_{0C})\}$  if  $G \in SEM(I)$ ,

Transition from initial state to former initial state providing that current graph satisfies initial graph class expression

$guard(G, s, x) = \{(G', s')\}$  if  $(G', s') \in guard_C(G, s, x)$ ,

Maintain transitions of the units' stepwise control

$guard(G, f_c, skip) = \{(G, f)\}$  if  $f_c \in F_C, G \in SEM(T)$ .

Transition from former final states to new final state providing that current graph satisfies terminal graph class expression

The computation of a stepwise control transformation unit is, according to the underlying unit, provided by Definition 87 (computation of basic stepwise control) or Definition 106 (computation of stepwise control for structured transformation unit).

# Chapter 8

## Conclusion

This chapter summarises the presented work and provides some suggestions for future work.

The thesis has introduced and examined three kinds of control conditions for transformation units, parallel expressions, as-long-as-possible expressions, and stepwise controls. Parallel expressions and as-long-as-possible expressions regard the expressiveness of control conditions. Both augment regular expressions by additional composition operators. The concept of stepwise control conditions focuses on the actual computation of semantics. The thesis has introduced basic stepwise controls, providing the basic components and computation of stepwise controls. Employing the concept of stepwise controls and modifying it when needed, the thesis implemented also parallel and downgraded as-long-as-possible expressions as stepwise control conditions.

Parallel expression is an umbrella term for three different kinds of expressions, weak parallel, proactive, and synchronous expressions, each of which employing another kind of parallel composition. Weak parallel composition enables but not demands the simultaneous application of rules. It allows the arbitrary composition of rules, simultaneously as well as sequentially. Proactive composition demands the simultaneous application of rules whenever it is possible, otherwise also sequential composition is allowed. Synchronous composition demands simultaneous application of the rules. If this is not possible, the synchronisation fails.

Considering weak parallel and synchronous expressions it was shown that these describe languages which turned out to be regular. This was proven by constructing finite state automata for weak parallel respectively synchronous expressions and then showing that the automata recognise exactly the respec-

tive languages. The language of a weak parallel or synchronous expression can be employed to construct a semantics consisting of all derivations whose rule application sequence is an element of the language. The thesis also introduced a second semantic approach based on a normal form of weak parallel and synchronous expressions. For every expression, the normal form provides the first (parallel) rule which has to be applied. Then in turns building the normal form, performing a derivation step, and again building the normal form for the remaining expression until the expression is entirely processed, permitted derivations can be build. Regarding future work, it would also be possible to construct a normal form, such that also proactive expressions could be taken into account. This normal form would not provide an already composed parallel rule, but give access to the single rules at choice to be applied proactively. Then the choice which of these rules are actually applied simultaneously is made according to the current graph, leaving the remaining rules to be applied later, i.e. these rules stay in the remaining expression which again is transformed to normal form.

The second sort of control conditions the thesis has introduced are as-long-as-possible-expressions. Adding an as-long-as-possible operator to regular expressions one is able to express the as long as possible execution of an entire, rather complex, process description. Examining as-long-as-possible expressions regarding termination the thesis introduced some sufficient conditions for termination based on the analysis of the expressions with the help of Petri nets. This approach is called assured termination. Thereby, the following is done for every subexpression  $C!$  of the considered as-long-as-possible expression. Every possible run through  $C$  is represented by a so called change vector, providing for each considered graph element to which amount it is added respectively deleted by the run. Thereby, the addition respectively deletion caused by subexpressions of the form  $\bar{C}^*$  or  $\bar{C}!$  is estimated by  $\infty$  respectively 0. This is a worst case estimation and it is left for future work to find better ones (if possible). Modelling the change vectors of the different runs as transitions and the kind of the considered graph elements as places of a Petri net, the interplay of the runs can be analysed employing analysing methods of Petri nets. If for all possible combinations of runs one place eventually runs out of tokens the subexpression  $C!$  terminates. And if this happens for all subexpressions  $C!$  of the considered as-long-as-possible expression, the entire expression terminates.

The third control condition introduced by the thesis is the stepwise control. The name stepwise control refers to the kind it is used to obtain permitted derivations. A basic stepwise control can be viewed as augmented finite state

automata. Being in a state it provides actions (rules) to be applied in the next derivation step, or regulation statements in order to steer the derivation process. In contrast to finite state automata it is able to take into account the current graph, or let its transitions depend on some other conditions. The execution of a basic stepwise control is implemented by computation steps on configurations, whereas a configuration represents the overall state of the stepwise control, i.e. for basic stepwise controls it comprises the current graph and its current state. In each computation step a transition is performed leading to a follower configuration. Carrying out computation steps until a final state is reached leads to a computation of the stepwise control. Given such a computation, one can easily "extract" a derivation, omitting all regulation steps. Hence, a stepwise control is able to build up only those derivations which may end up to be permitted.

We have straightforwardly modelled weak parallel and synchronous expressions as stepwise control, based on the construction of their finite state automata as these were introduced before. Being able to take into account the current graph, we are also able to construct stepwise controls for proactive expressions. Here, we pursue another approach implementing parallelism with stepwise controls. Instead of encoding the respective parallel expression directly by a stepwise control, now, the stepwise control only provides the parallel structure, i.e. it indicates which parts have to be (somehow) executed in parallel, but does not represent how. The actual weak, proactive, or synchronous composition of the provided rules is performed during computation.

We also modelled downgraded as-long-as-possible expressions as stepwise control. As a reminder, a downgraded as-long-as-possible expression,  $alap^-$  expression, is an as-long-as-possible expression without the operators  $*$  and  $|$ . An  $alap^-$  stepwise control indicates when an as-long-as-possible iteration begins and the computation then memorises the current graph, in order to continue with it in case the iteration fails. Therefore, the computation on configurations is modified. The configuration is augmented in order to include the graphs which have to be memorised. It now contains, besides the current graph and the current state also a sequence of graphs, whereas the leftmost graph of the sequence represents the graph currently present before the latest iteration. Whenever during computation a new as-long-as-possible iteration is indicated the graph currently present is memorised by adding it to the sequence. Whenever the current iteration fails the computation continues with the memorised graph which then is taken away from the sequence. For future work termination of an  $alap^-$  stepwise control could be examined. Being able to take into account the current graph could improve the ability to make estimations what is deleted respectively added by an

expression  $C$  which has to be iterated as long as possible. Pursuing such an approach postpones the question of (assured) termination to the moment of actual computation instead of checking beforehand. Another task could be to upgrade the  $alap^-$ -expression again. The choice operator  $|$  has been omitted, since the computation would have got too complex for the scope of the thesis. Taking into account the operator  $|$  again leads to the possibility to describe more complex behaviour. Moreover, a further task could be to construct stepwise controls for expressions which combine  $alap^-$  expressions with parallel expressions.

As a last topic we related transformation units and stepwise controls. At first we considered how to employ stepwise controls as control conditions for transformation units. Regarding simple transformation units one can straightforwardly use a stepwise control as control condition for the unit. Regarding structured units we have two possibilities to integrate the computation of the imported units into the computation of the calling unit. The first possibility is to treat an imported unit like a rule, i.e. its entire computation is embedded as one computation step in the computation of the calling unit, employing the "graph pair semantics" of the invoked unit. The second possibility is to embed the entire computation of an imported unit in a stepwise manner. This possibility requires the control conditions of the imported units to be stepwise controls. Moreover, we have to modify our computation approach in order to organise the executing of a calling unit and its imported units. We employ a stack of configurations, whereas each configuration represents a unit. The unit represented by the configuration on top of the stack is the one which is currently executed. Whenever an imported unit is invoked a new configuration representing this unit is pushed onto the stack and the computation continues with that unit. If its execution has terminated and its terminal graph class expression is satisfied the configuration is pulled from the stack and the computation continues with the underlying unit.

In order to employ a transformation unit itself as stepwise control, again the control condition of the unit has to be a stepwise control. This stepwise control is augmented in such a way that it provides only one transition from its start state, subject to the condition that the initial graph class expression of the underlying unit is satisfied. Analogously, a transition to a final state only is allowed if the terminal graph class expressions is satisfied. The computation stays the same as for structured units with stepwise controls. Being able to transform a transformation unit to a stepwise control, a suggestion for future work is to build one large stepwise control for a structured transformation unit, instead of organising the invocation of its imported units on

the level of computation. One could build a stepwise control from every imported unit and integrate them in the stepwise control of its calling unit. In order to do that, the control conditions of all units have to be decomposable and recomposable as stepwise control, e.g., regular expressions over rules and transformation units.

# Bibliography

- [Bau96] Bernd Baumgarten. *Petri-Netze. Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 1996.
- [BHPT05] Paolo Bottoni, Kathrin Hoffmann, Francesco Parisi-Presicce, and Gabriele Taentzer. High-level replacement units and their termination properties. *J. Vis. Lang. Comput.*, 16(6):485–507, 2005.
- [CMR<sup>+</sup>96] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation, part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1996.
- [DM78] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. Technical report, Stanford, CA, USA, 1978.
- [EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.
- [Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer Berlin Heidelberg, 1979.
- [EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Concurrency, Parallelism, and Distribution*. World Scientific Publishing Company, 1999.

- [EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, pages 167–180. IEEE Computer Society, 1973.
- [ER97] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 1–94. World Scientific Publishing Co., Inc., 1997.
- [Hab04] Annegret Habel. *Graphersetzungssysteme*. Lecture notes, 2004. [http://users.informatik.haw-hamburg.de/~klauck/GKA/Habel\\_skript.pdf](http://users.informatik.haw-hamburg.de/~klauck/GKA/Habel_skript.pdf).
- [HKK06] Karsten Hölscher, Renate Klempien-Hinrichs, and Peter Knirsch. Undecidable control conditions in graph transformation units. In Anamaria Moreira Martins and Leila Ribeiro, editors, *Brazilian Symposium on Formal Methods (SBMF 2006)*, pages 121–135, 2006.
- [HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [KK99] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.
- [KK07] Hans-Jörg Kreowski and Sabine Kuske. Autonomous units and their semantics – the parallel case. In *WADT'06: Proceedings of the 18th international conference on Recent trends in algebraic development techniques*, pages 56–73. Springer-Verlag, 2007.
- [KK11] Hans-Jörg Kreowski and Sabine Kuske. Graph multiset transformation – a new framework for massively parallel computation inspired by dna computing. *Natural Computing*, 10(2):961–986, 2011.
- [KK14] Hans-Jörg Kreowski and Sabine Kuske. *Theoretische Informatik I*. Lecture notes, University of Bremen, 2014.



- [KKK06] Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske. Some essentials of graph transformation. In *Recent Advances in Formal Languages and Applications*, pages 229–254. Springer, 2006.
- [KKR08] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph transformation units – an overview. pages 57–75, 2008.
- [KKS97] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1997.
- [Kre78] Hans-Jörg Kreowski. *Manipulationen von Graphmanipulationen*. PhD thesis, University of Berlin, 1978.
- [Kus98] Sabine Kuske. More about control conditions for transformation units. In *Proc. Theory and Application of Graph Transformations, volume 1764 of Lecture Notes in Computer Science*, pages 323–337, 1998.
- [Kus00] Sabine Kuske. *Transformation Units-A structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.
- [Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4), 1989.
- [Plu98] Detlef Plump. Termination of graph rewriting is undecidable. *Fundam. Inform.*, 33(2):201–209, 1998.
- [Ric08] Elaine Rich. *Automata, Computability, and Complexity – Theory and Applications*. Pearson Education, Inc., Upper Saddle River, New Jersey, USA, 2008.
- [Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
- [Sch92] Uwe Schöning. *Theoretische Informatik kurz gefasst*. BI-Wissenschaftsverlag, 1992.
- [VVE<sup>+</sup>06] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination analysis of model transformations by Petri nets. In *ICGT*, pages 260–274, 2006.