Networks of Rewriting Systems

M.Sc. thesis
by

Mihály Petreczky

Supervisor: Prof. dr. D. Janssens (University of Antwerpen)
Second reader: dr. F. van Raamsdonk

Vrije Universiteit
Amsterdam
2002
Contents

Preface iii

1 Introduction 1

2 Preliminaries 7
  2.1 Formal language theory ........................................... 7
  2.1.1 Basics from formal language theory ......................... 7
  2.1.2 Language processors ............................................ 9
  2.2 High-level replacement systems ................................. 11
  2.2.1 Graphs, graph morphisms and labeled graphs ............. 11
  2.2.2 High-level replacement productions and high-level re-
          placement systems (HLRS) ................................. 12
  2.2.3 Parallel application of high-level replacement productions 14

3 Networks of language processors 19
  3.1 Introduction ....................................................... 19
  3.2 CCNL systems ....................................................... 20
    3.2.1 General description of CCNL systems ..................... 20
    3.2.2 CCPC grammar systems .................................... 23
    3.2.3 Test-tube systems based on splicing ..................... 27
  3.3 PCLP systems ..................................................... 29
    3.3.1 General PCLP systems ..................................... 29
    3.3.2 PC grammar systems with Chomsky grammars as com-
          ponents ..................................................... 31

4 High-level replacement programs (HLRP) 33
  4.1 Introduction ...................................................... 33
  4.2 HLRP Categories ................................................... 33
  4.3 Derivation sequences in HLRP categories ..................... 40
    4.3.1 Basic notations ............................................. 40
    4.3.2 Construction of the "active" derivation sequence ....... 41
    4.3.3 Direct derivation equivalent to a derivation sequence .. 50
  4.4 High-Level Replacement Programs (HLRP) ..................... 52
    4.4.1 Syntax of high-level replacement programs ............... 53
4.4.2 Semantics of high-level replacement programs 55

5 Abstract networks of rewriting systems 59
5.1 Introduction 59
5.2 Abstract networks of rewriting systems 60
5.2.1 Category of ANRS states 60
5.2.2 Extension of syntax and semantics of high-level replacement programs 62
5.2.3 Abstract networks of rewriting systems 68
5.3 NLP systems as ANRS systems 70
5.3.1 Preliminary preparations 70
5.3.2 Language processors as ext-HLRP 80
5.3.3 CCNLP systems 82
5.3.4 PCLP systems 90

6 Conclusions and future work 97

A Properties of ANRS state categories 99
A.1 ANRS states as categories 99
A.2 Construction of limits and colimits 100
A.3 Inheritance of HLRP conditions 102
Preface

My first encounter with the broad area of networks of language processors took place in Hungary, while doing my fourth year of undergraduate studies at Eötvös Loránd University of Budapest. That time one of the leading researchers of this area, Dr. Erzsébet Csuhaj-Varjú, was giving a series of lectures on this subject at my alma mater. The freshness of the concept and the wide range of possible applications impressed me much, so I decided to study this field more profoundly. Erzsébet mentioned the need for a software environment which would be able to simulate various classes of networks of language processors. I liked the idea very much, but while thinking about the possibility to build up such an environment I ran into a problem: there was no uniform formalism for describing such systems. Clearly, it would be impossible to create a long-lived software environment without having such a formalism. So I started to look for one. Meanwhile following the advice of Erzsébet I read some works on actor grammars. The resemblance of actor systems to networks of language processors and the mathematical elegance of actor grammars impressed me much and inspired me to employ the concept of graph rewriting for describing networks of language processors in a uniform way. This work presents some results which have been achieved while working in this direction. In fact, to my great delight, I managed to create a uniform description I wished to. Of course, my plan to create a software environment hasn’t became true, but at least an important step has been made towards its realization.

There a lot of people I owe thanks to. Without any pretension to mention all of them I will try to make up a short list of those, who helped me very much in my work. First of all, I would like to thank Prof. Dirk Janssens. Despite his numerous engagements he has managed to find some time for me and agreed to be my supervisor. He helped me to master those concepts from the area of graph rewriting which I needed for this work. His broad knowledge of this area and his great experience were essential for carrying out the task. A great deal of the results of this thesis were inspired by his suggestions. Particularly, the material of Chapter 4 relies on the ideas he shared with me during consultations.

I would like to thank Dr. Erzsébet Cuhaj-Varjú for the moral and professional support she gave me. It was she who introduced me this field and helped me to make the first steps. Without her support I wouldn’t be able to write this work at all. I would like to thank Dr. Femke van Raamsdonk for her support and understanding. Whatever problem I had, she was always willing to help
me. She kindly agreed to be my second supervisor at the Vrije Universiteit and her advice influenced this work a great deal. I ought to thank Prof. Gregorz Rozenberg for his help. It was he who first gave me literature on graph rewriting and introduced me to Prof. Janssens. Despite his various obligations and duties Prof. Rozenberg had managed to find time for me, for what I am especially grateful. I would like to thank the family Vreeke and my neighbors for being my second family. Without the emotional comfort they created I wouldn’t be able to do as much as I did. I would like to thank my friend Vince Bárány for his support and his inspiring ideas. He was always willing to share his ideas and listen to mine. His broad knowledge and original views on basics of mathematics and computer science have enriched me a lot and helped me in my work.
Chapter 1

Introduction

Several phenomena of the surrounding environment can be modeled by complex systems made of several, usually simpler, components. These components, or agents, interact with each other, very often resulting quite complex behaviour. Such multi-agent systems are capable of carrying out complex computational tasks. The task to be accomplished is divided into several subtasks, each carried out by agents almost independently from each other. Since usually there are certain relationships between different subtasks, agents can’t work completely independently; they have to interact with each other in order to carry out the whole task.

The aim of this thesis is to present a generalization of certain formalizations of multi-agent systems. This generalization allows to describe a wide range of such systems in a uniform way.

The first step towards a formalization of multi-agent systems is to describe the computation process performed by agents. There are several ways to represent computation. A common approach is to view a computation process as a sequence consisting of several computation steps. Each computation step is identified with an application of a string operation. It is assumed that there are finitely many possible string operations. During the $i$th computation step the corresponding string operation is applied to the set (multiset) of strings which has been created by the $i - 1$th computation step (i.e. the computation step preceding the $i$th computation step). An initial set (multiset) of strings is assigned to each computation. The first computation step starts with this initial set (multiset) of strings. That is, work of each agent can be represented by a so called language processor. As the name suggests, a language processor is a formalism which determines sequences of applications of operations on strings depending on the set of strings to which the language processor has been applied. When applied to a set of strings, a language processor determines one or more possible sequences of string operations, which are to be applied to the input set of strings in the manner described above – i.e. each operation is applied to the result of the operation which precedes it in the sequence. In a broad sense, string grammars, term rewriting systems, Turing machines and
several other mathematical constructions can be viewed as language processors. In the light of the discussion above, a multi-agent system can be viewed as a systems consisting of several language processors. The communication of agents naturally corresponds to exchanging strings between agents. To sum up, agents correspond to language processors each having a separate pool of strings, work of agents is viewed as application of language processors to their pools of strings and interaction of agents is viewed as exchanging strings between string pools of the language processors. Such systems are called networks of language processors. From the description above it naturally arises that networks of language processors can be defined in terms of formal language theory. For more information on networks of language processors see [2, 8]. It is worth to note, that there are some other formal language theoretical approaches towards describing multi-agent systems, where agents are treated in the way described above, but their interaction is represented as changes made to the environment of the system. The environment is represented by a string, and each agent makes some changes to the environment by applying the associated language processor to a substring of the environment string. See [6, 20] for more information on these formalisms.

Note that in the description of multi-agent systems presented above the strings and string operations are merely used as a medium for storing data and encoding computation processes performed on that data. At the first glance that is a very natural approach. On the other hand, this is not the only possible way to model multi-agent systems. Data and computation can be represented by other mathematical structures as well. In many cases, such representations yield a more realistic model of the phenomenon to be described. A nice example might be formal models of DNA computers. One of the oldest models of DNA computers is the one based on splicing. In this case data are represented as strings and computation is represented as a series of string manipulations of a certain class, called splicing operations. Now a quite realistic model of DNA computers might be that one, where agents correspond to different test-tubes filled with some liquid containing DNA molecules, strings correspond to DNA molecules and computation corresponds to series of splicing rule applications. In this case communication is performed by exchanging DNA molecules, that is, strings, between test-tubes, that is, between agents. Such systems are called test-tube systems (see [5]). On the other hand, splicing operation is known to be difficult to implement. Some more promising models of DNA computers have been developed, where the underlying objects of computation (DNA molecules) are viewed as graphs. Therefore it might be desirable to develop models, where agents would perform computation not on strings or sets of strings but rather on graphs. Obviously graphs and manipulations on graphs are not the only possible alternative way to represent data and computation. A relatively young field of mathematics, called category theory, might help to create the desired level of abstraction. Namely, if we assume that data objects used by the agents form an instance of a certain class of mathematical structures – categories – we are able to define computational process on those data objects in a way which is very similar to rewriting of strings or graphs. Strings and graphs themselves
make up a category and the techniques used for categories reduce to classical string and graph rewriting in this case. The idea of describing data as objects of a category is crucial for this work. It provides us with sufficient level of abstraction for creating a uniform formalism for various kinds of multi-agent systems.

Another challenge is to describe the communication performed by agents. There lots of ways to specify how agents of a certain system should perform communication and it is far from being trivial to find a uniform formalism for representing communication. The way out of this situation which is suggested by this work is to encode communication by series of graph transformations. Graph transformations have been used for modeling communication of distributed multi-agent systems nearly for two decades. A remarkable and well studied approach towards representing multi-agent systems by graph transformation is that of actor grammars (see [18, 16, 17]). In this case states of the system are represented by graphs. Nodes of such graphs correspond to agents, edges correspond to different instances of interactions between agents (sending a message, being able to send a message and so on). In a quite similar way other aspects of communication could be encoded by graphs as well.

The approach introduced in the previous paragraph makes it possible to describe several aspects of communication. But one important problem remains to be solved: how to reflect those changes in the data objects of agents, which are caused by communication events? The solution presented here is to represent states of a multi-agent system as objects of a certain category. These objects have two parts: a communication graph, which encodes communication in the way described above and a data configuration part, which associates a data object with each agent. Then communication events are viewed as computational processes applied to the objects describing states of the system. The result of executing such a computational process coincides with result of a communication event.

Now we are almost ready to generalize the notion of network of language processors. We represent states as objects of a category of the type described above. We will call this category the category of state objects or simply the state category. We represent communication as performing a computation process on objects of the state category. We assign a data object to each agent and assume that data objects should be objects of a certain category. This category will be called the data object category of the system. We can describe the work of agents as computations performed on objects of the category of data objects. The only problem is that we would like describe the work of the system by means of transitions from one state of the system to another one. Assume now that we are interested only in systems, where either all agents work on their own data objects, or all agents take part in communication. We assume that is the latter case no agent changes its data object. Now in this case the work of the system can be described as a series of state transitions caused either by communication or by independent work of agents on their data objects. In the former case both the communication graph and the data configuration parts of the object encoding the state of the system are changed, in the latter case just the data
configuration part is changed. For characterizing the work of the system in the manner presented above, one needs to define how to perform computation in parallel manner. Besides, one needs to specify how to reflect the result of a computational process applied to data objects on the data configuration part of the corresponding state object.

To do this, computation performed on objects of a certain category will be defined in two steps. First, the definition of rewriting rules on objects of a category, analogous to rewriting of strings or graphs, should be given. These rules are called \textit{high-level replacement productions}. Then, constructions prescribing order of applications of these rules are to be defined. These constructions are called \textit{high-level replacement programs}. Computation performed on a certain object can be defined as applying a certain high-level replacement program to that object. We assign a high-level replacement program to each agent. The effect of the event when all agents work independently on their data objects will be simulated in the following way. For each agent the program associated with that agent will be applied to the data object of the agent. The effect of a communication event corresponds to applying a high-level replacement program to the object which encodes the current state of the system.

In the following chapters the ideas sketched above will be developed. That is, such multi-agent systems will be formalized which work in two alternating phases: a communication phase, in which no agent works on its data object and some agents communicate with each other and a phase in which all agents work on their data objects without any interaction with each other. The main lines of the formalization are listed below.

- Data objects which the agents of the system act upon are assumed to form a category.
- A high-level replacement program acting on the level of data objects is assigned to each agent.
- A data object is associated with each agent.
- A high-level replacement program is defined over objects encoding the state of the system implements communication. Each communication step is realized by applying this program to the object encoding the state of the system.
- The phases in which all agents are working on their data objects are implemented by parallel application of programs associated with agents. For each agent the program assigned to that agent is applied to the data object of the agent.

Systems described in this manner will be called \textit{abstract networks of rewriting systems}. Here the word "abstract" indicates that the term denotes a rather general notion. The phrase "network of rewriting systems" reflects the fact that we have a generalization of networks of language processors where agent essentially work in the same way as in the case of networks of language processors,
but instead of string manipulations, operations on more abstract objects are allowed. Since manipulations of data objects occur by performing programmed rewriting of objects, the usage of the term "rewriting systems" for denoting agents' activities is justified.

Note that the model above not only describes the work of a certain multi-agent system by simulating the possible state transitions of the system, but also captures the dynamics of the system. By allowing parallel execution of high-level replacement programs, both the phase when all agents work on their data objects and the phase when agents interact with each other are simulated in a way which preserves the dynamics. Computations performed independently by agents are modeled by parallel execution of programs in such a way that each program which is executed simulates the work of exactly one agent. Besides, the communication of agents can be simulated by high-level replacement programs in such a way that independent stages of communication are executed in parallel. That is, the framework of abstract networks of rewriting systems embraces more than just the possible state transitions of multi-agent systems. It also contains knowledge about the dynamic behaviour of multi-agent systems.

The structure of this work is the following. Chapter 2 contains some preliminaries on formal language theory and rewriting defined on objects of categories. Chapter 3 gives some basic models of networks of language processors. Chapter 4 introduces the notion of programmed rewriting of objects of a category. Chapter 5 introduces the notion of abstract network of rewriting systems and demonstrates the expressive power of this formalism by describing systems from Chapter 3 within the framework of abstract networks of rewriting systems. Appendix A contains some technical proofs of statements from Chapter 5. Chapter 6 contains some conclusions and hints about future development of the concepts introduced in this work.

The material of Chapter 3 and Chapter 2 is taken from the literature. The material of Chapter 4 and Chapter 5 - to author's knowledge - hasn't been described in the literature yet.
Chapter 2

Preliminaries

The aim of this chapter is to briefly describe notions and facts needed to understand the following chapters. Section 2.1 contains some notions from theory of formal languages necessary to understand Chapters 3 and 5. For comprehensive survey on the theory of formal languages see [23, 22]. Section 2.2 contains a short summary of definitions and facts about high-level replacement systems and graph rewriting. For more on this topic see [13, 24] In the latter section it is assumed that the reader is familiar with basics of category theory. There are a lot of good textbooks on category theory, see for example [1].

2.1 Formal language theory

In this section some notions and notations from the theory of formal languages will be described. Subsection 2.1.1 deals with those basic notions of formal language theory which are necessary to understand the next chapter. For a profound survey on formal language theory see [22, 23]. In Subsection 2.1.2 the notion of language processors will be defined. This is an essential notion of the whole work. More on this topic can be found in [2]

2.1.1 Basics from formal language theory

Finite sets of symbols are called alphabets. Alphabets will be denoted by capital Latin letters. Strings of symbols from a certain alphabet $V$ will be called strings over alphabet $V$. Strings will be denoted by small Greek and Latin letters. If $x = a_1 a_2 \ldots a_n$ and $y = b_1 b_2 \ldots b_m$ are strings over a certain alphabet $V$ that is, $a_i, b_j \in V$ ($1 \leq i \leq n, 1 \leq j \leq m$), then the concatenation of $x$ and $y$ is the string $xy = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m$. The empty string will be denoted by $\epsilon$. The empty string has the following property: for each string $x$ we have that $x \epsilon = \epsilon x$. Strings over a certain alphabet $V$ with concatenation as binary operation on strings form a semigroup with neutral element. The neutral element is the empty string. The length of a string $x = a_1 a_2 \ldots a_n$ where $a_i \in V$ ($1 \leq i \leq n$)
is denoted by $|x|$ and equals $n$ in this case. That is, the length of a string is the number of symbols of the string. If $K \subseteq V$ then $|x|_K := \{1 \leq i \leq n | a_i \in K\}$. That is, $|x|_K$ is the number of those symbols in $x$, which belong to $K$.

By a language over a certain alphabet we mean a set of strings over that alphabet. There are a lot of possible ways to define languages, one of them is to give constructive methods to generate strings of the language. Such constructions are usually called language generating devices. The usual approach is to define a finite set of rewriting rules which change strings and a finite set of initial strings with which the rewriting procedure starts. Then a set of strings obtained from the set of initial strings by applying rewriting rules determines a language.

A string of the form $p \rightarrow q$ such that $\rightarrow \notin V$ and $p, q \in V^*$ and $p \neq \epsilon$ will be called a Chomsky-grammar rewriting rule. The left-hand side of a rule $p \rightarrow q$ is $p$, the right-hand side of the rule is the string $q$. Each such a rewriting rule defines a binary relation on strings. This binary relation will be denoted by $\Rightarrow_{p \rightarrow q}$ and is defined for each $x, y \in V^*$ in the following way:

$$x \Rightarrow_{p \rightarrow q} y \text{ if and only if } x = uv \land y = uqv$$

Let $P$ be a finite set of rewriting rules of the form $p \rightarrow q$. Then $\Rightarrow_P$ is the relation:

$$\Rightarrow_P = \bigcup_{p \rightarrow q \in P} \Rightarrow_{p \rightarrow q}$$

By $\Rightarrow_P^*$ we will denote the reflexive transitive closure of $\Rightarrow_P$.

One of the oldest and most important language generating devices are classical Chomsky-grammars. The formal definition is the following.

**Definition 2.1.1.** A Chomsky-grammar is a 4-tuple $G = (N, T, S, P)$ where

- $N$ is the alphabet of non-terminal symbols
- $T$ is the alphabet of terminal symbols
- $S \in N$ is the starting symbol (the axiom) of the grammar.
- $P$ is a finite set of rewriting rules. The rules are of the form $p \rightarrow q$ where $p, q \in V^*$ and $p$ contains at least one non-terminal symbol: $|p|_N > 0$.

The language generated by the grammar $G$ is the set $L(G) = \{w \in T^* | S \Rightarrow_P^* w\}$

A Chomsky-grammar is called context-free if the left-hand sides of its rewriting rules are strings containing only one symbol.

Splicing is another important class of operations defined on strings. The idea comes from some standard laboratory techniques employed in molecular biology. A splicing rule is a string of the form $p\#q\$u\$v$ such that $p, q, u, v \in V^*$ and
# Formal Language Theory

Given a splicing rule \( r = p#q#u#v \) for \((x,y),(z,w) \in V^* \times V^*\) we say that \((z,w)\) can be derived from \((x,y)\) using the splicing rule \( r \) if and only if

\[
x = x_1pqx_2, \quad y = y_1uvy_2, \quad z = x_1pqy_2, \quad w = y_1uqvx_2.
\]

We denote the fact that \((z,w)\) is derivable from \((x,y)\) by using splicing rule \( r \) by \((x,y) \vdash_r (z,w)\).

**Definition 2.1.2.** A splicing scheme is a 2-tuple \( \sigma = (V,R) \) where \( V \) is a finite alphabet and \( R \) is a finite set of splicing rules over \( V \). Let \( L \subseteq V^* \)

- \( \sigma(L) = \{ w \in V^*| \exists x, y \in L, z \in V^*: (x,y) \vdash_r (w,z) \text{ for certain } r \in R \} \)
- \( \sigma^0(L) = L \)
- \( \sigma^{i+1}(L) = \sigma(i)(L) \cup \sigma(\sigma(i)(L)) \)
- \( \sigma^*(L) = \bigcup_{k \geq 0} \sigma^k(L) \)

A splicing scheme \( \sigma \) can be viewed as a language generating device. Given a finite set \( A \) of strings over \( V \), the language generated by \( \sigma \) might be defined as \( \sigma^f(A) \) where \( f = *, 0, 1, 2, \ldots \)

## 2.1.2 Language processors

In fact, both Chomsky-grammars and splicing schemes described above are particular cases of a more general formal construction called language processor. This general notion was introduced in [2] and covers a big class of language generating devices. The formal definition goes as follows.

**Definition 2.1.3.** A language processor \( \Pi \) is a 3-tuple \( \Pi = (V,M,P) \), where \( V \) and \( M \) are disjoint finite alphabets, \( P \subseteq (V \cup M)^+ \), \( V \) is the alphabet of the language processor \( \Pi \), \( M \) is the alphabet of marker symbols, \( P \) is the finite set of rewriting rules of \( \Pi \).

In the definition above the way how rewriting rules should be applied is not defined explicitly. Instead, a rewriting rule is coded as a string consisting of symbols of the alphabet and marker symbols. The latter is intended to be used for defining the structure of a rewriting rule.

Let \( \Pi \) be a language processor as defined in Definition 2.1.3 and let \( A \subseteq V^* \) be a set of strings over the alphabet \( V \). To define the language generated by the language processor \( \Pi \) the following notations will be used:

- Denote by \( \Pi(A) \) the set of strings obtained from \( A \) by applying rewriting rules from \( P \).
- The following notations are intended to denote iterated applications of rules from \( P \) to elements of \( A \). \( \Pi^i(A) \) is going to denote the set of strings obtained from \( A \) by rewriting \( k \) times \((0 \leq k \leq i)\) elements of \( A \). In case
of \( k = 0 \) it means that the elements of \( A \) are left unchanged. Formally it looks like this:

\[
\begin{align*}
\Pi^0(A) &= A \\
\Pi^{i+1}(A) &= \Pi(\Pi^i(A)) \cup \Pi^i(A) \\
\Pi^*(A) &= \bigcup_{i=1}^{\infty} \Pi^i(A)
\end{align*}
\]

By \( \Pi^*(A) \) we denote the set of normal forms with respect to \( P \) of those strings, which belong to \( A \). That is, \( \Pi^*(A) \) is the set of all those strings which can be derived from a string in \( A \) by successive application of rules from \( P \) but can’t be rewritten by any rules from \( P \). That is:

\[
\Pi^*(A) = \Pi^*(A) \cap \{ w \in V^* | \Pi(w) = \emptyset \}
\]

If \( A \) is a multiset of strings then \( \Pi^f(A) \) where \( f = \ast, \ast, 0, 1, \ldots \) can be defined in the similar manner. Symbols \( f = \ast, \ast, 0, 1, \ldots \) are called rewriting modes. We will associate a rewriting mode \( f = \ast, \ast, 0, 1, \ldots \) with each language processor \( \Pi \). We will reflect it by saying that a particular language processor \( \Pi \) works in the rewriting mode \( f \), where \( f \) can be \( \ast, \ast, 0, 1, \ldots \). Rewriting modes determine the language generated by a language processor in the following way:

**Definition 2.1.4.** Consider a language processor \( \Pi = (V, M, P) \) which works in rewriting mode \( t = \ast, \ast, 0, 1, 2, \ldots \). The language generated by \( \Pi \) starting from a finite set of initial axioms \( A \subseteq V^* \) is the language \( \Pi^t(A) \).

Note that a language processor \( \Pi \) may generate different languages while working in different rewriting modes.

To illustrate the definitions introduced above, two examples will be given below. The first one gives the representation of a Chomsky-grammar as a language processor, the second one gives the representation of splicing schemes in terms of language processors.

**Example 2.1.1.** A Chomsky-grammar \( G = (N, T, S, P_G) \) might be viewed as a language processor \( \Pi = (N \cup T, \{ \to \}, P_G) \). The language processor \( \Pi \) rewrites a set of strings \( A \) in the following way: \( \Pi(A) = \{ w \in V^* | \exists u \in A : u \Rightarrow_{P_G} w \} \). The language generated by \( G \) is \( L(G) = \Pi^* \{ S \} \cap T^* \). Define the rewriting mode \( t \) as \( \Pi^t(A) = \Pi^*(A) \cap T^* \). Then \( L(G) = \Pi^t \{ S \} \), that is, the language processor \( \Pi \) while working in the rewriting mode \( t \) generates the language \( L(G) \) if started from the set \( \{ S \} \).

**Example 2.1.2.** A splicing scheme \( \sigma = (V, P_\sigma) \) might be viewed as a language processor \( \Pi = (V, P_\sigma) \). The language generated by the language processor \( \Pi \) starting from set of strings \( A \) while working in rewriting mode \( f = \ast, 0, 1, 2, \ldots \) coincides with \( \sigma^f(A) \). That is, \( \Pi^f(A) = \sigma^f(A) \) where \( f = \ast, 0, 1, 2, \ldots \) and \( A \subseteq V^* \).
2.2 High-level replacement systems

Rewriting of strings, terms and graphs are well known areas which have been studied for several decades. For the purposes of this work a more general notion of rewriting is needed. Namely, we would like to rewrite not just strings or graphs but objects of an arbitrary category. To do this, a generalization of the double-pushout approach (DPO for short) for graph rewriting will be used. The rewriting systems described in this section are known as high-level replacement systems. They were introduced in [13] and further developed in [24]. The DPO approach itself is described in several papers, see for example [12, 11]. Subsection 2.2.2 deals with the notion of high-level replacement systems and high-level replacement productions. Subsection 2.2.3 contains some results on parallelism in high-level replacement systems: the definition of parallel application of high-level replacement productions together with some basic results will be given there. Concepts described in Subsection 2.2.2 and Subsection 2.2.3 will be illustrated by examples for the category of labeled graphs and the category of sets. The category of labeled graphs together with related notions and notations will be described in Subsection 2.2.1. For more information on the category of labeled graphs and on rewriting of labeled graphs see [13, 12, 11, 24].

2.2.1 Graphs, graph morphisms and labeled graphs

A directed finite graph $G$ is a 4-tuple $G = (V_G, E_G, t_G, s_G)$ where $V_G$ and $E_G$ are finite sets and $t_G, s_G : E_G \to V_G$ functions from $E_G$ to $V_G$. The set $V_G$ is called the set of nodes of the directed graph $G$, $E_G$ is called the set of edges of the directed graph $G$. For each edge $e \in E_G$ the node $t_G(e)$ is the target node of $e$ and $s_G(e)$ is the source node of the edge $e$. Wherever it doesn’t lead to ambiguity, the word graph rather than the expression "directed finite graph" will be used. Let $G = (V_G, E_G, t_G, s_G)$ and $H = (V_H, E_H, t_H, s_H)$ be graphs. A graph morphism $h : G \to H$ is a 2-tuple $h = (h_V, h_E)$ such that $h_V : V_G \to V_H$, $h_E : E_G \to E_H$ and $t_H(h_E(e)) = h_V(t_G(e))$ and $s_H(h_E(e)) = h_V(s_G(e))$. The graph morphism $h$ is said to be injective (surjective) if $h_V$ and $h_E$ are injective (surjective). A graph morphism is isomorphism if it is injective and surjective. Graph $G$ is said to be a subgraph of $H$ if $V_G \subseteq V_H$, $E_G \subseteq E_H$ and $t_H|_{E_G} = t_G$ and $s_H|_{E_G} = s_G$. The fact that $G$ is a subgraph of $H$ will be denoted by $G \subseteq H$. Note that graphs with graph morphisms form a category. A labeled graph $G$ over label alphabet $Lab = Lab_e \cup Lab_v$ ($Lab_e \cap Lab_v = \emptyset$) is a tuple $G = (V_G, E_G, t_G, s_G, l_v^G, l_e^G)$ where $(V_G, E_G, t_G, s_G)$ is a directed graph and $l_v^G : E_G \to Lab_v, l_e^G : E_G \to Lab_e$ are the labeling functions. Let $G$ and $H$ be labeled graphs. A label preserving graph morphism $h : G \to H$ is a graph morphism $(h_V, h_E) : (V_G, E_G, t_G, s_G) \to (V_H, E_H, t_H, s_H)$ such that $l_H \circ h_V = l_v^G$ and $l_H \circ h_E = l_e^G$. A label preserving graph morphism is said to be injective (surjective) if it is injective (surjective) as a graph morphism. We will denote by $\text{LGraph}$ the category which has labeled graphs as objects and label preserving graph morphisms as arrows.
2.2.2 High-level replacement productions and high-level replacement systems (HLRS)

Consider an arbitrary category $\text{CAT}$ and a class $M$ of arrows of the category $\text{CAT}$. The definition of rewriting rules and application of rewriting rules over objects of $\text{CAT}$ will be given below. To make the rather abstract definitions easier to understand examples for concrete categories $\text{Set}$ and $\text{LGraph}$ will be given. Here $\text{Set}$ denotes the category of sets with functions as arrows and sets as objects. See [13, 1] for more information on category $\text{Set}$. In the case of category $\text{Set}$ let $M$ be the class of injective set-to-set functions. In the case of category $\text{LGraph}$ take the class of injective graph morphisms as the class of arrows $M$.

**Definition 2.2.1.** A high-level replacement production $p = (L \xrightarrow{\longleftarrow} K \xrightarrow{r} R)$ consists of objects $L$ - the left-hand side of the production, $R$ - the right-hand side of the production, $K$ - the gluing object and arrows $l : K \rightarrow L$ and $r : K \rightarrow R$ which belong to the class $M$.

For the category $\text{Set}$ a production rule consists of three sets $L, K$ and $R$ and two injective functions: $l : K \rightarrow L$ and $r : K \rightarrow R$. In the category $\text{LGraph}$ of labeled graphs a production rule consists of three labeled graphs $L, K$ and $R$ and two injective label preserving graph morphisms: $l : K \rightarrow L$ and $r : K \rightarrow R$.

**Definition 2.2.2.** Given an object $G$, a production $p = (L \xrightarrow{\longleftarrow} K \xrightarrow{r} R)$ and a morphism $o : L \rightarrow G$ the production $p$ is applicable to $G$ through morphism $o$ if and only if there exist such an object $C$ and an object $H$ and arrows $h, k, g, f$ that diagrams (1) and (2) below are pushouts:

```
        L   K   R
        ↓    ↓    ↓
o   o   h   f
  G    C    H
```

The morphism $o : L \rightarrow G$ is called an occurrence morphism. $H$ is said to be derived from $G$ via production $p$ and the occurrence morphism $o$. This is denoted by $G \Rightarrow_{p,o} H$. Object $C$ is called the pushout complement object.

Let $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ be a high-level replacement production. Let $\Rightarrow_p$ denote the following binary relation on objects of $\text{CAT}$:

$$G \Rightarrow_p H \text{ if and only if } \exists o : L \rightarrow G : G \Rightarrow_{p,o} H$$

Let $P$ be a finite set of high-level replacement productions. Then define $\Rightarrow_P$ in the following way: $\Rightarrow_P = \bigcup_{p \in P} \Rightarrow_p$. The reflexive transitive closure of $\Rightarrow_P$ will be denoted by $\Rightarrow_P^*$. 

2.2. **HIGH-LEVEL REPLACEMENT SYSTEMS**

Since categories \textbf{Set} and \textbf{LGraph} have all finite limits and colimits, from which it follows that these categories have all pushouts, the applicability of a production rule to an object is equivalent to the existence of a pushout complement object. Construction of pushouts in category \textbf{Set} and \textbf{LGraph} are described in several papers, see [13, 12, 11].

In the category \textbf{Set} a production rule \( L \xleftarrow{f} K \xrightarrow{g} R \) is applicable to an object \( G \) through \( o : L \to K \) if and only if the following condition holds:

\[
\{ m \in L | \exists n \neq m : o(m) = o(n) \} \subseteq l(K)
\]

Condition (2.1) is called \textit{gluing condition} for sets.

In category \textbf{LGraph} a production rule \( L \xleftarrow{f} K \xrightarrow{g} R \) is applicable to \( G \) through \( o : L \to G \) if and only if the following two conditions are satisfied:

**Identification condition**

\[
\{ m \in E_L \cup V_L | \exists n \neq m : o(n) = o(m) \} \subseteq l(K)
\]

**Dangling condition**

\( \forall e \in E_G : (t_G(e) \in o(L) \land s_G(e) \in G \setminus o(L)) \implies t_G(e) \in o(l(K)) \)

and

\( \forall e \in E_G : (s_G(e) \in o(L) \land t_G(e) \in G \setminus o(L)) \implies s_G(e) \in o(l(K)) \)

The two conditions above are known as \textit{gluing conditions} for labeled graphs.

**Definition 2.2.3.** A sequence \( G_0 \Longrightarrow_{p_1,o_1} G_1 \Longrightarrow_{p_2,o_2} \ldots G_{n-1} \Longrightarrow_{p_n,o_n} G_n \) is a \textit{derivation sequence} if either \( n = 0 \) or for each \( i = 1 \ldots n \) \( G_i \Longrightarrow_{p_i,o_i} G_i \) holds.

Let \( P \) be a finite set of productions. Then \( G \Longrightarrow^*_P H \) holds if and only if there exists a derivation sequence \( G_0 \Longrightarrow_{p_1,o_1} G_2 \Longrightarrow_{p_2,o_2} \ldots G_{n-1} \Longrightarrow_{p_n,o_n} G_n \) with \( n \geq 0 \), \( G_0 = G \) and \( G_n = H \) and \( p_i \in P \) for all \( i = 1 \ldots n \).

**Definition 2.2.4.** A high-level replacement system is a 3-tuple \( \text{HLRS} = (S, P, T) \) where \( S \) is an object of \( \text{CAT} \) – called the axiom of the system, \( P \) is a finite set of productions, \( T \) is a class of objects – the class terminal objects. \( L(\text{HLRS}) = \{ t \in T | S \Longrightarrow^*_P t \} \) is the \textit{language} generated by the high-level replacement system \( \text{HLRS} \).

High-level replacement systems are a generalization of rewriting systems defined on graphs or strings. A rewriting system over labeled graphs is usually defined as a 3-tuple \( G = (S_G, P_G, T_G) \). Here \( S_G \) is a labeled graph called the axiom of the system \( G \), \( P_G \) is a finite set of rewriting rules for labeled graphs. \( T_G \) is a set of node and edge labels called the set of terminal labels. The language \( L(G) \) is simply the set of all those graphs which are labeled exclusively by terminal labels and are derivable from \( S \) using production rules from \( P \). The corresponding high-level replacement system looks like this: \( \text{HLRS}(G) = (S_G, P_G, T) \) where \( T \) is the set of all graphs labeled by labels from \( T_G \)
2.2.3 Parallel application of high-level replacement productions

High-level replacement rules are a suitable tool for generalizing rewriting systems with parallel application of rules. To do this the notion of parallel and sequential independence is needed. Parallel and sequential independent derivations were defined first for labeled graphs (see [12, 11]). Here the definition for high-level derivations will be given.

Definition 2.2.5. Let \( p = (L \xrightarrow{r} K \xrightarrow{l} R) \) and \( p' = (L' \xrightarrow{l'} K' \xrightarrow{r'} R') \) be high-level replacement productions. Direct derivations \( G \Longrightarrow_{p,o} H \) and \( G \Longrightarrow_{p',o'} H' \) are said to be parallel independent if and only if the following diagram commutes:

\[
\begin{array}{c}
R & \xrightarrow{r} & K & \xrightarrow{l} & L \\
| & & | & & |
\downarrow{(2)} & & \downarrow{(1)} & & \downarrow{o} \\
C & \xrightarrow{o'} & G & \xleftarrow{o'} & C' \\
| & & | & & |
\uparrow{G} & & \uparrow{H} & & \uparrow{H'} \\
H & \leftarrow{p,o} & L & \leftarrow{l'} & K' & \leftarrow{r'} & R'
\end{array}
\]

where \((1) + (2)\) and \((1') + (2')\) are diagrams of derivations \( G \Longrightarrow_{p,o} H \) and \( G \Longrightarrow_{p',o'} H' \) respectively. That is, diagrams \((1), (2), (1'), \) and \((2')\) are pushouts.

In the case of labeled graphs two direct derivations are parallel independent if and only if the intersection of the images of the left-hand sides are equal to the intersection of the images of the gluing objects: \( o(l(K)) \cap o'(l'(K')) = o(L) \cap o'(L') \).

Definition 2.2.6. Let \( p = (L \xrightarrow{l} K \xrightarrow{r} R) \) and \( p' = (L' \xrightarrow{l'} K' \xrightarrow{r'} R') \) be high-level replacement production rules. Direct derivations \( G \Longrightarrow_{p,o} H \) and \( H \Longrightarrow_{p',o'} X \) are said to be sequentially independent if and only if the following
2.2. **HIGH-LEVEL REPLACEMENT SYSTEMS**

Diagram commutes:

![Diagram](image)

where (1) + (2) and (1') + (2') are diagrams of derivations $G \Rightarrow_{p,o} H$ and $H \Rightarrow_{p',o'} H'$ respectively. That is, diagrams (1), (2), (1') and (2') are pushouts.

For labeled graphs the definition above is equivalent to the condition that the intersection of the image of the right-hand side of production $p$ with the image of the left-hand side of production $p'$ equals to the intersection of the images of the gluing objects: $h(R) \cap o'(L') = h(r(K)) \cap o'(l'(K'))$.

It is easy to see that if we take the category of labeled directed graphs then for each two sequential independent direct derivations $G \Rightarrow_{p,o} H$ and $H \Rightarrow_{p',o'} X$ it is possible to construct a direct derivation $G \Rightarrow H'$ with the same effect as derivation sequence $G \Rightarrow_{p,o} H \Rightarrow_{p',o'} H'$. Here the expression "the same effect" means that the same components of graph $G$ are deleted and the same components are added. If we take two parallel independent derivations as described in Definition 2.2.5 then $L'$ can be applied to $H$ through $L' \rightarrow C \rightarrow H$ and direct derivations $G \Rightarrow_{p,o} H$ and $H \Rightarrow_{p',L' \rightarrow C \rightarrow H} X$ are sequentially independent. On the other hand, it is possible to construct a derivation $G \Rightarrow_{p+p'} X$ with the same effect as $G \Rightarrow_{p,o} H \Rightarrow_{p',L' \rightarrow C \rightarrow H} X$.

Here $p+p'$ denotes the production $(L+L' \vdash_{a+b} K+K' \vdash_{r+r'} R+R')$ where $a+b$ stands for the coproduct of objects $a$ and $b$ which is the disjoint union of $a$ and $b$ in the case of labeled graphs and sets. That is, $p+p'$ corresponds to parallel composition of $p$ and $p'$ and the derivation $G \Rightarrow_{p+p'} X$ corresponds to parallel application of $p$ through $o$ and $p'$ through $o'$.

This relationship between sequential and parallel independent direct derivations is a well known fact for labeled graphs. The same connection between sequential and parallel independent direct derivation can be proven for high-level replacement productions over an arbitrary category CAT with class of arrows $M$ provided that the category $CAT$ and the class of arrows $M$ satisfies certain conditions. These conditions are called HLR1 conditions and categories which satisfy these conditions will be called HLR1 categories in this work.
**Definition 2.2.7.** A category $\mathbf{C A T}$ together with class $M$ of morphisms is said to satisfy *HLRI conditions* (see [13]) if and only if each of the following conditions holds:

**Existence of pushouts and pullbacks for $M$-morphisms**

- Let $A, B, C$ be objects of $\mathbf{C A T}$ and let $A \to B$ and $A \to C$ be morphisms belonging to the class $M$. Then the pushout of the arrows $A \to B$ and $A \to C$ exists with some pushout object $D$. That is, the diagram (1) below is a pushout diagram.

\[
\begin{array}{ccc}
A & \longrightarrow & B \\
\downarrow & & \downarrow \\
C & \longrightarrow & D
\end{array}
\]

- Let $E, F, G$ be objects of $\mathbf{C A T}$. Let $E \to G$ and $F \to G$ be morphisms belonging to class $M$. Then the pullback of arrows $E \to G$, $F \to G$ exists with some pullback object $H$. That is, the diagram (2) below is a pullback:

\[
\begin{array}{ccc}
H & \longrightarrow & E \\
\downarrow & & \downarrow \\
F & \longrightarrow & G
\end{array}
\]

**Inheritance of $M$-morphisms under pushout and pullback**

- In the pushout diagram (1) above, if $A \to B$ belongs to $M$ then it implies that $C \to D$ belongs $M$.

- In the pullback diagram (2) above, if $E \to G$ belongs to $M$ and $F \to G$ belongs to $M$ then $H \to F$ and $H \to E$ belong to $M$.

**Triple pushout condition** Consider the following diagram:

\[
\begin{array}{ccc}
A & \longrightarrow & B \\
\downarrow & \downarrow & \downarrow \\
A' & \longrightarrow & C & \longrightarrow & D \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
B' & \longrightarrow & D' & \longrightarrow & E
\end{array}
\]
2.2. **HIGH-LEVEL REPLACEMENT SYSTEMS**

Assume that morphisms $A \to B$, $A' \to B'$, $C \to D$, $C \to D'$, $D' \to E$ and $D \to E$ belong to $M$. Then

(a) if $(1) + (2)$ and $(1') + (2)$ are pushouts and $(2)$ is a pullback then $(1)$, $(2)$ and $(1')$ are pushouts.

(b) If $(2)$ is a pushout, then $(2)$ is a pullback too.

**Existence of Binary Coproducts** Category $C A T$ has all binary coproducts.

For all morphisms $f : A \to A'$ and $g : B \to B'$ that belong to $M$, their coproduct morphism $f + g : A + B \to A' + B'$ belongs to $M$ too.

Sometimes it is convenient to use the result of the following lemma to prove that a certain category and class of morphism satisfies HLR1 conditions.

**Lemma 2.2.1.** Assume that $C A T$ satisfies all HLR1 conditions except the Triple Pushout Condition. Consider the following diagram:

```
\begin{diagram}
  & A & \rightarrow & B \\
  & \downarrow & & \downarrow \\
  C & \rightarrow & D' & \rightarrow & D \\
  \downarrow & & \downarrow & & \downarrow \\
  G & \rightarrow & \phantom{D'} & \rightarrow & \phantom{D} \\
\end{diagram}
```

Assume that for all such diagrams the following is true. If morphisms $A \to B$, $C \to D'$, $D' \to D$ and $C \to G$ are in $M$, $(1) + (2)$ is a pushout and $(2)$ is a pullback, then $(1)$ and $(2)$ are pushouts. Then the category $C A T$ with class of arrows $M$ satisfies Triple Pushout Condition too.

If the category $C A T$ and class of arrows $M$ satisfies the HLR1-conditions, then the following theorem holds:

**Theorem 2.2.1.** Let $G \Rightarrow_p H$ and $G \Rightarrow_p H'$ be parallel independent direct derivations. Then there exist direct derivations $H \Rightarrow_{p'} X$ and $H' \Rightarrow_p X$ such that derivations $G \Rightarrow_p H \Rightarrow_{p'} X$ and $G \Rightarrow_{p'} H' \Rightarrow_p X$ are sequentially independent and there exists a derivation $G \Rightarrow_{p + p'} X$ such that $p + p' = (L + L' \leftarrow K + K' \rightarrow R + R')$, where + denotes the coproduct of two objects.

The derivation $G \Rightarrow_{p + p'} X$ can be viewed as the result of parallel application of $p$ through $o$ and $p'$ through $o'$.

The theorem can generalized for more than two derivations.

**Corollary 2.2.1.** Let $G \Rightarrow_{p_i \cdot o_i} H_i$ be parallel independent derivations ($i = 1 \ldots n$). Then there exists an object $X$ such that:
(i) $G \Rightarrow \sum^n_{i=1} p_i X$

(ii) for all permutations $\pi \in \text{perm}(1, \ldots, n)$:

$$G \Rightarrow_{p_{\pi(1)}} D_{\pi(1)} \Rightarrow_{p_{\pi(2)}} D_{\pi(2)} \cdots \Rightarrow_{p_{\pi(n)}} D_{\pi(n)} = X$$

and $D_{\pi(1)} = H_{\pi(1)}$ and the first derivation of the sequence is the derivation

$G \Rightarrow_{p_{\pi(1)}} H_{\pi(1)}$

By using Lemma 2.2.1 it is easy to see that

- Category $\text{CAT} = \text{Set}$ and $M =$ class of injective set-to-set functions

- Category $\text{CAT} = \text{LGraph}$ with $M =$ class of injective morphisms

satisfy the HLR1 conditions. This means that the results stated above are true for sets and labeled graphs.
Chapter 3

Networks of language processors

3.1 Introduction

In this chapter some basic models of networks of language processors (NLP for short) will be dealt with. NLP systems are built of agents, each having a repository of strings. Each agent is identified with a language processor. Agents rewrite the contents of their string repositories and communicate with each other by exchanging strings. For detailed overview of NLP systems see [2, 3, 8]. Along with more general description some important particular cases of NLP systems, such as PC grammars, CCPC grammars, test-tubes systems will be treated in this chapter. For more details on these systems see [7, 15, 4, 9, 5]. A on-line bibliography of this area can be found at http://www.sztaki.hu/mm/bib.html

Work of such systems is usually described as an alternating sequence of two steps: a rewriting step and a communication step. During a rewriting step each agent performs rewriting of its strings. During a communication step, agents exchange their strings and store them in their repositories. Usually synchronous work is assumed, in a sense that either all agents are participating in a rewriting step or all agents are participating in a communication step. Of course, some agents may be inactive during any of these steps. New agents might be allowed to appear and existing agents might disappear during the work of NLP systems. Generally, if we think of agents as nodes of a graph and communication between agents as edges between nodes corresponding to agents taking part in communication then we get the so called communication graph of the system. According to what has been written above the communication graph of a NLP system is allowed to change during the process of computation performed by the system.

Usually networks of language processors are considered as language generating devices. There are several ways to associate set of strings to a sequence of computation steps. One very common approach is the following: fix a certain
agent, the *master agent*, and define the language generated by the particular NLP as a subset of strings which appear in the repository of the master agent. In most of cases only those strings are considered to belong to the language generated by the particular NLP which contain only symbols over a certain alphabet (called the *terminal alphabet*).

NLP systems might be classified by the communication protocol used by the agents of the system, the dynamics of the communication graph, the way how strings are rewritten by agents and so on. In this chapter two important subclasses of NLP systems will be treated: *networks of language processor communicating by command* (CCNLp) and networks of language processor communicating by request. The latter version will be referred to as *systems of parallel communicating language processors* (PCLP).

The main difference between CCNLp and PCLP systems is the way how agents of the system communicate with each other. In the case of CCNLp systems, each rewriting step is followed by a communication step. In the case of PCLP systems, a communication step takes place only if some agents require it. Notice that by introducing dummy communication steps, which simply check whether there are any requests from agents for communication and perform a communication step if needed, the work of PCLP systems can be viewed as alternating sequence of communication and rewriting steps. For both types of NLP systems only the case with fixed number of agents will be treated in this chapter.

A very profound and comprehensive survey of CCNLp systems can be found in [2, 3]. For more information on PCLP systems see [8].

### 3.2 CCNLp systems

In this section the general description of CCNLp systems will be given together with definition of two particular cases: CCPC grammars and test-tube systems based on splicing. These particular examples are not only a nice illustration of the concept of CCNLp systems but they are quite important and well studied models as well.

#### 3.2.1 General description of CCNLp systems

The notion of CCNLp as described below was introduced in [2]. Agents are doing rewriting on strings, each agent has a repository of strings. This repository may be treated either as a set or as a multiset. Each agent has a set of rewriting rules, an *input filter* and an *output filter*. Filters are languages, which specify which strings may enter and which strings may leave the repository of each agent. During a rewriting step, each agent changes its strings by using its rewriting rules. During a communication step, strings from one agent’s repository are copied to other agent’s repository. Only those strings can leave the ith agent’s repository, which are members of the ith agent’s output filter. Only those strings are allowed to enter the ith agent’s repository, which are members of the ith agent’s
3.2. **CCNL Systems**

Input filter. Communication takes place according to the *neighborhood relation*. For the *ith* agent the neighborhood relation gives the set of those agents, which the *ith* agent is allowed to communicate with. This set is called the set of neighbors of the *ith* agent. That is, an agent communicates only with its neighbors. The *jth* agent gets all those strings sent by its neighbors, which are able to leave their repositories and are able to pass the input filter of the *jth* agent. Notice that if all agents are allowed to communicate with all other agents, then we get broadcast communication. CCNL systems with broadcast communication are of great importance, since some of the most prominent representatives of CCNL systems belong to them. Particularly, CCPC grammars, test-tube systems are CCNL systems communicating in broadcast manner. Such CCNL systems will be referred to as CCBNL systems. In this model the number of agents doesn’t change in the process of computation. Test-tube systems and CCPC grammars are particular instances of the CCNL model. A configuration of the model is a tuple, the *ith* element of which is the content of the *ith* agent’s repository. The languages generated by a CCNL system is the set of those strings, which appear in the repository of a certain agent, called the *master agent*, during the computation started from a certain initial configuration.

Now we are ready to give a formal definition of CCNL systems

**Definition 3.2.1.** A CCNL system $\Gamma$ is a structure $\Gamma = (V, M, s_0, A_1, \ldots, A_n, R)$ where:

- $V$ is the finite alphabet of the system
- $M$ is the finite alphabet of marker symbols of the system and $M \cap V = \emptyset$.
- $A_i$ ($i = 1 \ldots n$) are the components or agents of the system. Components are of the form $A_i = (I_i, O_i, \Pi_i, f_i)$ where:
  - $\Pi_i = (V_i, M_i, R_i)$ is a language processor, $V_i \subseteq V$, $M_i \subseteq M$.
  - $I_i \subseteq V_i^*$ is the input filter of the *ith* component
  - $O_i \subseteq V_i^*$ is the output filter of the *ith* component
  - $f_i = \ast', \ast, 0, 1, 2, \ldots$ is the rewriting mode of the language processor $\Pi_i$.
- $s_0 = (S_1, \ldots, S_n)$ is the initial state of the systems. Here for each $i = 1 \ldots n$
  - $S_i \subseteq V_i^*$ is a finite set (multiset). $S_i$ is called the set of initial axioms of the *ith* component.
- $R \subseteq Ag \times Ag$ the neighborhood relation. Here $Ag = \{A_1, \ldots, A_n\}$.

**Remark 3.2.1.** In the literature the term "component" rather then the term "agent" is used. In the subsequent text expressions "component" and "agent" will be used interchangeably.

**Remark 3.2.2.** If $R = Ag \times Ag$ the we have CCNL systems with broadcast communication. Such systems will be referred to as CCBNL systems.
In each moment in time, the state of the system Γ is described by a $n$-tuple of sets (or multisets) of strings $(L_1, \ldots, L_n)$, where $L_i$ is a set (or multiset) of strings over $V_i$. Both rewriting step and communication step can be viewed as transition from one state of the system to another state. The transition relation defined by rewriting steps will be denoted by $\implies_{rew}$. The transition relation defined by communication steps will be denoted by $\implies_{comm}$. Assume that the master agent is the $j$th one. Then the language generated by the CCNLP system $\Gamma$ is the following set:

$$L_{NLP}(\Gamma) = \{ w \in V^* | (s_1, \ldots, s_n) \implies_{rew} (x_1^{(1)}, \ldots, x_n^{(1)}) \implies_{comm} (x_1^{(2)}, \ldots, x_n^{(2)}) \implies_{rew} \ldots \implies_{rew} (x_1^{(k)}, \ldots, x_n^{(k)}) \land k \geq 0 \land w = x_j^{(k)} \}$$

Below the formal definitions of relations $\implies_{rew}$ and $\implies_{comm}$ will be given.

The effect of a rewriting step can be described as follows:

$$(L_1, \ldots, L_n) \implies_{rew} (L_1', \ldots, L_n') \text{ if and only if } L_i' = \Pi_i^\beta(L_i) \text{ for all } i = 1 \ldots n$$

The formalization of a communication step is slightly more complicated. There are various way to define a communication step performed by a CCNLP system depending on the communication protocol used by the particular class of CCNLP systems. Here it will be defined formally for three particular cases, which are still general enough:

**Protocol (a)** String objects are distributed among agents according to their input and output filters. String objects, which are not able to join any agent from the neighborhood or are not able to leave their original agent remain in the repository of their original agent.

**Protocol (b)** For each agent all string objects of the agent remain in the agent’s repository. Copies are made of each string objects and these copies are distributed among agents according to their input and output filters.

**Protocol (c)** Similar to the protocol (a), but strings which can’t leave their agents or can’t join any other agent in the neighborhood are discarded, besides strings are not allowed to enter their agent’s repository, even if they would pass the input and output filter of the agent.

To reflect the dependence of communication steps on the communication protocol, $\implies_{comm}^{prot}$ will be used instead of $\implies_{comm}$ to denote the transition relation determined by communication steps. Here $prot \in \{ (a), (b), (c) \}$.

**Definition 3.2.2.** $(L_1, \ldots, L_n) \implies_{comm}^{prot} (L_1', \ldots, L_n') \text{ (} prot \in \{ (a), (b), (c) \} \text{)}$ if and only if depending on the value of $prot$ one of the following holds:

- Protocol (a)
  
  $\forall i = 1 \ldots n : L_i' = L_i \cap (\bigcup_{1 \leq j \leq n \land j \notin E(R(A_i))} O_j \cap L_i) \\
  \cup (L_i \setminus ((L_i \cap O_i) \cap (\bigcup_{1 \leq j \leq n \land j \notin E(A_i)} I_j)))$
3.2. CCNL P SYSTEMS

- Protocol (b)

\[ \forall i = 1 \ldots n : L_i' = L_i \cup (I_i \cap ( \bigcup_{1 \leq j \leq n \land j \neq i} (O_j \cap L_j))) \]

- Protocol (c)

\[ \forall i = 1 \ldots n : L_i' = I_i \cap ( \bigcup_{1 \leq j \leq n \land j \neq i} (O_j \cap L_j)) \]

Examples of such systems are CCPC grammars and test-tube systems based on splicing or cut/recombination rules. The notion of CCNL P systems is rather general, which makes extremely difficult to find meaningful results for general CCNL P systems. Therefore most of known results were formulated for certain special classes of CCNL P systems.

3.2.2 CCPC grammar systems

A CCPC grammar system is such a CCBNLP system whose agents have just one string at each rewriting/communication step and agents have no output filters. Each agent has a single symbol as its starting axiom and a set of rewrite rules. The input filter is called the selector language in this case. A particular class of CCPC grammars are those with regular languages as selector languages, Chomsky grammars as their rewriting rules, and with each agent doing rewriting step as long as possible. CCPC grammars were intended as models of certain classes of parallel machine architectures, such as the WAVE paradigm, Connection Machine and Boltzmann machine. More on CCPC grammars can be found in [7, 15]

Definition 3.2.3. A CCPC grammar system is a CCBNLP system \( \Gamma = (N \cup T, s, \{\rightarrow\}, A_1, \ldots, A_n) \) such that:

- \( N \) and \( T \) are disjoint alphabets, \( T \) is the set of terminal symbols, \( N \) is the set of nonterminal symbols
- \( s = (S_1, \ldots, S_n) \) such that \( S_i \in N \) for all \( i = 1 \ldots n \). \( S_i \) is the axiom of the \( i \)th agent.
- \( A_i \) is of the form \( A_i = (R_i, \emptyset, (N \cup T, P_i), \star) \) where \( R_i \subseteq (N \cup T)^* \) is the selector language of the \( i \)th component and \( (N, T, S_i, P_i) \) is a Chomsky grammar.

For CCPC grammars the master agent (or the master component) is the first one. Formally a rewriting step looks like this:

\[ (x_1, \ldots, y_n) \rightarrow_{\text{rew}} (y_1, \ldots, y_n) \text{ if and only if} \]

\[ \forall i = 1 \ldots n : x_i \rightarrow_{P_i}^* y_i \text{ and no rule from } P_i \text{ can be applied to } y_i \]
In the literature CCPC grammars are defined as tuples
\[ \Gamma = (N, T, (S_1, P_1, R_1), \ldots, (S_n, P_n, R_n)) \]
where \( N \), \( T \), \( P_i \), \( R_i \), \( S_i \) are the same as in the definition above. The communication steps are different from any determined by protocols (a), (b), (c) formally described in the previous section. Four types of CCPC grammars are distinguished depending on the communication protocol used by agents of the system.

- If during communication steps each agent resets its string to its initial axiom provided that no string has passed its input filter but its original string has passed the input filter of at least one agent, then the system is called returning.
- If in the case described above the agent continues to work on its original string, then the CCPC system is called non-returning.
- If during communication steps the whole string is sent, then the system is said to perform communication without splitting.
- If during a communication step strings are split into several substrings and substrings are distributed among agents, then the system is said to perform communication with splitting.

First the formal definition of the communication steps in the case of communication without splitting will be given.

In the case of communication without splitting each agent sends its whole string to other agents. After the communication step is over, the string of the \( i \)th agent changes according to the following rules:

- If there exists a non-empty string sent by some other agent which has passed the input filter of the \( i \)th agent, then the concatenation of such strings will be the new string of the \( i \)th agent.
- If no non-empty string has passed the input filter of the \( i \)th agent and the original string of the \( i \)th agent hasn’t passed the input filter of any other agent, then the string of the \( i \)th agent remains unchanged.
- If no non-empty string has passed the input filter of the \( i \)th agent but the original string of the \( i \)th agent managed to pass the input filter of some other agent, then there are two possible cases: either the system is returning and in this case the \( i \)th agent string will be its initial axiom, or the system is non-returning and the string of \( i \)th agent remains unchanged.

Formally it looks like this. Let \((x_1, \ldots, x_n)\) denote the current state of the system. Let
\[
\sigma(x_i, j) = \begin{cases} 
\epsilon & \text{if } x_i \notin R_j \text{ or } i = j \\
x_i & \text{otherwise}
\end{cases}
\]
\[ \delta(i) = \sigma(x_1, i) \sigma(x_2, i) \ldots \sigma(x_n, i) \]
\[ \sigma(i) = \sigma(x_i, 1) \sigma(x_i, 2) \ldots \sigma(x_i, n) \]

Notice that \( \sigma(i) = \epsilon \) if and only if either the string of the \( i \)th agent is the empty string or the string of the \( i \)th agent hasn’t passed the input filter of any other agent. Similarly, \( \delta(i) = \epsilon \) if and only if there is no non-empty string sent by other agents which has passed the input filter of the \( i \)th agent. Using these notations a communication step for systems with communication without splitting can be formulated as follows: \((x_1, \ldots, x_n) \xrightarrow{\text{comm}} (y_1, \ldots, y_n)\) if and only if

\[ y_i = \begin{cases} 
\delta(i) & \text{if } \delta(i) \neq \epsilon \\
x_i & \text{if } \delta(i) = \epsilon \text{ and } \sigma(i) = \epsilon \\
S_i & \text{if } \delta(i) = \epsilon \text{ and } \sigma(i) \neq \epsilon \text{ and the system is returning} \\
x_i & \text{if } \delta(i) = \epsilon \text{ and } \sigma(i) \neq \epsilon \text{ and the system is non-returning}
\end{cases} \]

for all \( i = 1 \ldots n \)

In the case of communication with splitting not the entire string of an agent but a substring of it is sent to other agents. The communication goes as follows. Only those agents are allowed to send their strings whose strings can be represented as concatenation of strings belonging to input filters of the system. Such agents are called active components. An active component sends the substring of its string belonging to the input filter of the \( j \)th agent to the \( j \)th agent. All other agents are just allowed to accept messages (i.e. strings) from the active components. The string of the \( i \)th agent changes in the following way

- If there exists a non-empty string which has passed the input filter of the \( i \)th agent, then the concatenation of such strings is going to be the new string of the \( i \)th agent.
- If no non-empty string has passed the input filter of the \( i \)th agent and the \( i \)th agent is not an active component, then its string remains unchanged.
- If no non-empty string has passed the input filter of the \( i \)th agent and the \( i \)th agent is an active component, then two cases are possible: either the system is a returning one and in this case the new string of the \( i \)th agent is its initial axiom, or the system is a non-returning one and the string of the agent remains unchanged.

Formally it looks like this.

\((x_1, \ldots, x_n) \xrightarrow{\text{comm}} (y_1, \ldots, y_n)\)

if and only if there exists a set \( M \subseteq \{1, 2, \ldots n\} \) (set of active components) such that:

- \( \forall i \in M(\exists! \pi_i \in \text{perm}(n) : x_i = x_{1, \pi_i(1)}x_{2, \pi_i(2)} \ldots x_{n, \pi_i(n)}) \)

\( \wedge x_{i,i} = \epsilon \land \forall k \in 1 \ldots n : (k \neq \pi_i^{-1}(i)) \land (x_{i, \pi_i(k)} \in R_{\pi_i(k)} \lor x_{i, \pi_i(k)} = \epsilon) \)
\[ \forall i \in \{1, 2, \ldots, n\} \setminus M : \forall j \in \{1, \ldots, n\} : x_{i,j} = \epsilon \]

\[ \forall j \in 1, \ldots, n : y_j = \begin{cases} 
  x_{1,j} \ldots x_{n,j} & \text{if } x_{1,j}x_{2,j} \ldots x_{n,j} \neq \epsilon \\
  x_j & \text{if } (-\exists i \in M : x_{i,j} \in R_j) \land j \notin M \\
  S_j & \text{if } (-\exists i \in M : x_{i,j} \in R_j) \land j \in M \\
  x_j & \text{if } (-\exists i \in M : x_{i,j} \in R_j) \land j \in M \
  \text{and the system is returning} \\
  \end{cases} \]

The language generated by CCPC grammar system \( \Gamma \) is the language \( L_{\text{CCPC}}(\Gamma) = L_{\text{NLP}}(\Gamma) \cap T^* \). That is, it contains all strings over the terminal alphabet \( T \) which appear in the first agent’s repository after performing an alternating sequence of rewriting and communication steps.

**Example 3.2.1.** The following example is a returning CCPC grammar communicating without splitting. The grammar system is of the form:

\[
\Gamma = (N,T,(S_1, P_1, R_1), (S_2, P_2, R_2), (S_3, P_3, R_3)) \\
N = \{S_1, S_2, S_3, X, S'_2, S'_3\} \\
T = \{a, b, c\} \\
P_1 = \{S_1 \to aS_1, S_1 \to bS_1, S_1 \to X\} \\
R_1 = \{a, b\}^*c \\
P_2 = \{S_2 \to S'_2, X \to c\} \\
R_2 = \{a, b\}^*X \\
P_3 = \{S_3 \to S'_3, X \to c\} \\
R_3 = \{a, b\}^*X 
\]

A derivation sequence starting from the initial state \((S_1, S_2, S_3)\) goes like this:

\[(S_1, S_2, S_3) \implies_{\text{rew}} (xX, S'_2, S'_3) \implies_{\text{comm}} (S_1, xX, xX) \]

\[(x \implies_{\text{rew}} (wX, xc, xc) \implies_{\text{comm}} (xcx, wX, wX) \implies_{\text{rew}} (xcx, wX, wX) \]

where \(x, w \in \{a, b\}^*\). Note that \((xcx, wX, wX)\) is a terminal state, no state transition is possible from this state. Consequently, the language generated by the system \(\Gamma\):

\[L(\Gamma) = \{xcx|x \in \{a, b\}^*\}\]
3.2. CCNLP SYSTEMS

3.2.3 Test-tube systems based on splicing

In the case of test-tube systems, agents are identified with test-tubes, containing
set of biomolecules. Infinite supply of biomolecules is assumed. Each agent has
a set of rewriting rules. The rewriting rules are splicing rules. Agents have only
input filters. Each input filter is given by a finite alphabet and consists of words
over that alphabet. A profound study of test-tube systems with splicing can
be found in [5]

Test-tube systems can be viewed as formal language theoretical models of
would-be DNA computers. Systems similar to test-tube systems with splicing
but using different string operations have been studied too. Results proven for
test-tube systems are of great importance since they show that it is possible to
create DNA computer with universal computational power. Although currently
researchers are mainly focused on quite different models of DNA computers, test-
tube systems with various kinds of string operations performed on the contents
of tubes are still popular topic among theoretical computer scientists.

The formal definition of test tube systems is the following:

**Definition 3.2.4.** A test tube system is a CCBNLP system of the form \( \Gamma = \langle V, \{\$, \#\}, s_0, A_1, \ldots, A_n \rangle \) where each agent is of the form

\[
A_i = (V_i^*, \emptyset, \sigma_i, \ast)
\]

where \( V_i \subseteq V \) and \( \sigma_i = (V, \{\$, \#\}, R_i) \) is a splicing scheme. The initial state is

\[
s_0 = (s_1, \ldots, s_n) \text{ with } s_i \subseteq V^* \text{ for each } i = 1 \ldots n.
\]

Each agent works till no further rewriting is possible on the strings of the
agent. If one agent finishes earlier, then it waits until all agents have finished
rewriting their strings. In fact, such a definition of the rewriting step comes from
the biological motivation of the model. The model was created as a formalization
of a DNA computer which uses several test-tubes. Each test tube contains DNA
strands which are changed (rewritten) according to the specific rules of the test-
tube. It was reasonable to assume infinite supply of strings. That is, during
rewriting infinite copies of new strings evolve, while infinite number of copies of
strings from the original content of the test-tube do remain in the liquid. After
no further changes are possible (no change resulting a new string) contents of the
tubes are poured together, then using filters split apart in a way, that each test-
tube gets only DNA strands matching certain pattern(s). The communication
of the system follows the communication protocol (a). That is:

\[
(L_1, \ldots, L_n) \Rightarrow (L'_1, \ldots, L'_n) \text{ iff } \forall i \in [1, n] :
\]

\[
L'_i = \bigcup_{1 \leq j \leq n} (\sigma^*_j(L_j) \cap V_i^*) \cup (\sigma^*_i(L_i) \cap B), \text{ where } B = V^* - \bigcup_{1 \leq i \leq n} V_i^*
\]

Observe that relation \( \Rightarrow \) is the composition of relations \( \Rightarrow_{\text{comm}} \circ \Rightarrow_{\text{rew}} \) where \( \Rightarrow_{\text{comm}} \) is defined as for CCNLP using protocol (a) and \( \Rightarrow_{\text{rew}} \) is defined
as for CCNLP systems with splicing schemes as language processors assigned to the agents of the system. In the literature a slightly different formulation is used. Besides, only communication steps are considered as state transition steps. Below the formal definition which is used in the literature will be given.

**Definition 3.2.5.** A test tube system \( \Gamma \) is a tuple

\[
\Gamma = (V, (V_1, R_1, A_1), \ldots, (V_n, R_n, A_n))
\]

where:

- \( V \) is a finite alphabet
- \((V_i, R_i, A_i) \ (1 \leq i \leq n) \) is a component of the system, \( V_i \subseteq V \) is the selector language of the component, \( \sigma_i = (V, R_i) \) forms a splicing scheme and \( A_i \subseteq V^* \) is the axiom of the \( i \)th component.
- \( n \)-tuple \((L_1, \ldots, L_n)\) of languages over \( V \) is called the configuration of the system. Language generated by test-tube system \( \Gamma \) is defined in the following way:

\[
L_\Gamma = \{ w \in V^* | (A_1, \ldots, A_n) \Rightarrow^* (L_1, \ldots, L_n) \land w \in L_1 \}
\]

Observe that if we assume that the first agent is the master agent then the language \( L_\Gamma \) coincides with the language generated by \( \Gamma \) considered as a CCBNLP system.

**Example 3.2.2.** Consider the following test-tube system:

\[
\Gamma = \{ (a, b, c, d), \{ \{ abc, ebd, dae \}, \{ b^{\#} e^{\#} c^{\#}, da^{\#} e^{\#} c^{\#} a \}, \{ a, b, c \}, \{ \{ ec, ce \}, \{ b^{\#} d^{\#} e^{\#} c, c^{\#} d^{\#} a \}, \{ a, b, d \} ) \}
\]

A derivation from the initial state goes as follows:

\[
(\{ abc, ebd, dae \}, \{ ec, ce \} ) \Rightarrow_{rew} (\{ abc, ebd, dae, cab^{2} d, ec, da^{2} bc, ec, da^{2} b^{2} d \}, \{ ec, ce \} ) \Rightarrow_{com} (\{ abc, ebd, dae, cab^{2} d, ec, da^{2} bc, ec \}, \{ da^{2} b^{2} d, ec, ce, da^{2} b^{2} d \}) \Rightarrow_{rew} (\{ abc, ebd, dae, cab^{2} d, ec, da^{2} bc, ec, da^{2} b^{2} d \}, \{ ec, ce, da^{2} b^{2} d, da^{2} b^{2} d, \ldots, ca^{2} b^{2} d, de, ed, da^{2} b^{2} c \}) \ldots
\]

In the first tube, only rewriting of the following type occurs:

\[
(aa^{i} b^{j} \alpha, ebd) \Rightarrow (aa^{i} b^{j+1} d, ec)
\]

\[
(ca^{i} b^{j} \alpha, dae) \Rightarrow (da^{i+1} b^{j} \alpha, ce)
\]

where \( \alpha \in c, d \). In the second test tube, only rewriting of the following type might occur:

\[
(aa^{i} b^{j} d, ec) \Rightarrow (aa^{i} b^{j} c, ed)
\]
Strings of the form $da^i b^j c$ are in normal form w.r.t. the splicing rules of the first test-tube. Strings $ca^i b^j c$ are in normal form w.r.t. the splicing rules of the second test-tube. Besides only strings of the form $da^i b^j d$ can be passed to the second tube, and only strings $ca^i b^j c$ can go from the second tube to the first one. Besides all strings in the first tube are of the form $da^i b^j d$ and such strings are transformed by the second tube to strings of the form $ca^i b^j c$. So the language generated by system $\Gamma$ has the property that $L(\Gamma) \cap a^i c^j a^k = \{ a^n b^n c | n \geq 1 \}$.

### 3.3 PCLP systems

In general form PCLP systems were first described in [8, 2]. In this section first the general notion will be introduced, then – as a particular example – PC grammar systems with Chomsky grammars as agents will be described. More on PC grammar systems can be found in [4, 9, 10]

#### 3.3.1 General PCLP systems

A general description of PCLP systems can be found in [8]. A PCLP system consist of a fixed number of agents. We will only treat PCLP systems with each agent having a single string in its repository. Each agent has a set of string rewriting rules. In fact, agents can be viewed as language processors acting on a single string. Each agent has an axiom – the string which is in the agent’s repository at the beginning of the computation. The system has a set of special symbols, called query symbols. During a rewriting step each agent does a series of rewriting steps on its string. As soon as a query symbol appears in an agent’s string the system starts a communication step. During a communication step each query symbol $Q_i$ is substituted by the current string of the $i$th agent. This can be done in several ways. A PCLP system is returning, if every agent, after sending its string, continues rewriting on its axiom. A PCLP system is non-returning, if after sending its string each agent continues working on its string. PCLP systems with subword communication have been described in the literature too. In the case of subword communication not the entire string of an agent but just a substring of it is sent. Such systems have been studied in [10]

A configuration of the system is a tuple containing the current strings of agents. The initial configuration of the system is the tuple containing axioms of the agents. The language generated by a PCLP system is usually defined as a certain subset of strings which appear at the master agent at a certain step of computation. In many cases strings of the generated language are required to contain only symbols of the terminal alphabet.

Unlike for CBPNL systems, the communication here is not a broadcast communication: the $i$th agent may get strings only from those agents it has required from.

PCLP might be viewed as language theoretical models of parallel computer architectures. Another interpretation is to consider them as language theoreti-
cal models of multi-agent systems that use classroom model of problem solving. Language generating power of different classes of PCLP depending on complexity of components, number of components, mode of communication have been studied for several years (nearly a decade). Several interesting results were found, some of them do shed a light on the computational power of parallelism.

The formal definition looks like this.

**Definition 3.3.1.** A system of parallel communicating language processors is a n + 1 tuple \( \Gamma = (V, K, (S_1, \ldots, S_n), \Pi_1, \ldots, \Pi_n) \) where

- \( V \) is the finite alphabet of the system
- \( \Pi_i = (V, M, \nu, f_i) \) is a language processor working in mode \( f_i \). The language processor \( \Pi_i \) corresponds to the \( i \)th agent of the system.
- \( (S_1, \ldots, S_n) \) is the initial configuration of the system. \( S_i \in V^* \) is the axiom of the \( i \)th agent.
- \( K = \{Q_1, \ldots, Q_n\} \) is the alphabet of query symbols, containing exactly \( n \) elements, one for each agent.

**Remark 3.3.1.** In the literature the term ”component” rather then the expression ”agent” in used. In the subsequent text these two word will be used interchangeably.

States of the system can be described by \( n \)-tuples of strings. Transitions between states occur in the following manner:

- \( (x_1, \ldots, x_n) \rightarrow_{\text{trans}} (y_1, \ldots, y_n) \) if and only if
\[ \forall i \in \{1, \ldots, n\} : |x_i|_K = 0 \land y_i \in \Pi_i(f_i(x_i)) \]

- There exists an \( 1 \leq i \leq n \) such that \( |x_i|_K \neq 0 \). In this case a communication step takes place. The formal description will be given for case of PCLP systems with such a way of communication that the sender sends its entire string not just a substring of that. Three cases might be distinguished depending on the communication protocol used. \( (x_1, \ldots, x_n) \rightarrow_{\text{comm}} (y_1, \ldots, y_n) \) if and only if:
  - (a) For all \( i = 1 \ldots n \) if \( |x_i|_K \neq 0 \) and \( x_i = z_1 Q_{i_1} z_2 Q_{i_2} \ldots z_k Q_{i_k} z_{k+1} \) and \( 1 \leq j \leq k : |x_{i_j}|_K = 0 \) then \( y_i = z_1 x_{i_1} z_2 \ldots z_k x_{i_k} z_{k+1} \) and for each \( 1 \leq j \leq k \) let be \( y_{i_j} = x_{i_j} \) if the system \( \Gamma \) is non-returning or \( y_{i_j} = S_{i_j} \) if the system is returning. For all other cases not described above \( y_i = x_i \).
  - (b) For all \( i = 1 \ldots n \) if \( |x_i|_K \neq 0 \) and \( x_i = z_1 Q_{i_1} z_2 Q_{i_2} \ldots z_k Q_{i_k} z_{k+1} \) then \( y_i = z_1 u_i z_2 \ldots z_k u_i z_{k+1} \) where
\[ \forall 1 \leq j \leq k : u_j = \begin{cases} x_j & \text{if } |x_j|_K = 0 \\ Q_{i_j} & \text{otherwise} \end{cases} \]
3.3. PCLP SYSTEMS

\[ y_j = \begin{cases} 
  x_{i_j} & \text{if } |x_{i_j}|_K = 0 \text{ and the system } \Gamma \text{ is non-returning} \\
  S_{i_j} & \text{if } |x_{i_j}|_K = 0 \text{ and the system } \Gamma \text{ is returning} 
\end{cases} \]

and for all other cases not described above \( y_i = x_i \).

- (c) Consider \( Q_j \) such that there exists \( |x_k|_{Q_j} \neq 0 \) and \( |x_j|_{Q_j} = 0 \), \((1 \leq j, k \leq n)\). For each \( 1 \leq i \leq n \) if \( |x_i|_{Q_j} \neq 0 \) and \( x_i = z_1 Q_j z_2 Q_j \cdots z_k Q_j z_{k+1} \) then \( y_i = z_1 x_j z_2 \cdots z_k x_j z_{k+1} \) and

\[ y_j = \begin{cases} 
  x_j & \text{if the system is non-returning} \\
  S_j & \text{if the system is returning} 
\end{cases} \]

For all other cases not described above \( y_i = x_i \).

Let be \( \Rightarrow \Rightarrow \Rightarrow \rightarrow_{rew} \cup \Rightarrow\rightarrow_{comm} \). The language generated by the PCLP system \( \Gamma \) is the set

\[ L(\Gamma) = \{ w_1 \in T^*| (S_1, \ldots, S_n) \Rightarrow (w_1, \ldots, w_n) \} \]

A PCLP system gets blocked in the following cases:

- If it is not possible to execute a rewriting step for a component’s string.
- If during the communication step no queries can be fulfilled, that is, for all strings containing query symbols replacement of the query symbols by the strings of the corresponding agents is not possible.

A circular query is a particular example of configurations which lead the system to getting blocked. A circular query occurs when the \( i_k \)th component’s string contains query symbol \( Q_{k+1} \), for \( 1 \leq k < m \leq n \) and the string of the \( i_m \)th component contains query symbol \( Q_{i_k} \). In this case queries can not be fulfilled, and after a certain number of steps the system gets blocked.

3.3.2 PC grammar systems with Chomsky grammars as components

PC grammar systems is a well studied subclass of PCCL systems. PC grammar systems have vast literature, for results and various modifications see [4, 8, 10].

Here the language processors are Chomsky grammars. The alphabet of the system is the union of two disjoint sets: the set of terminal symbols and the set of non-terminal symbols. Formally such a PC grammar system can be defined like this.

**Definition 3.3.2.** A **PC grammar system** with \( n \) components is a tuple:

\[ \Gamma = (N, T, K, G_1, \ldots, G_n) \]

where:

- \( T \) is the set of terminal symbols
\( N \) is the set of non-terminal symbols
\( K = \{Q_1, \ldots, Q_n\} \) is the set of query symbols
\( G_i = (N, T, S_i, P_i) \) is the Chomsky-grammar of the \( i \)th component.

Using the formalism of Definition 3.3.1 a PC grammar system with Chomsky-grammars as components is a PCLP system \( G = (V, K, \Pi_1, \ldots, \Pi_n) \) with \( V = N \cup T, \Pi_i = (N \cup T, \{\to\}, P_i, t) \) where \( \Pi_i \) is a language processor of type described in Example 2.1.1. Communication takes place in the same way as described in Subsection 3.3.1. In [10] PC grammars with subword communication were investigated. Probably the most remarkable result on PC grammar systems with Chomsky-grammars as components is that PC grammar systems with 11 context-free components generate all recursively enumerable languages while working according to the communication protocol (a).

Example 3.3.1. Consider the returning PC grammar system \( \Gamma \) which performs communication steps according to protocol (a):

\[
\begin{align*}
\Gamma &= \{\{S_1, S_2, S_3\}, \{a, b, c, d\}, K = \{Q_1, Q_2, Q_3\}, G_1, G_2, G_3\} \\
P_1 &= \{S_1 \to aS_1, S_1 \to aQ_2, S_3 \to d\} \\
P_2 &= \{S_2 \to bS_2, S_2 \to bQ_3\} \\
P_3 &= \{S_3 \to cS_3\}
\end{align*}
\]

The derivation might go in several ways:

- \( (S_1, S_2, S_3) \Rightarrow \text{rew} \ (a^iS_1, b^iQ_3, c^iS_3) \Rightarrow \text{comm} \ (a^iS_1, b^iS_3, S_3) \)

The derivation gets blocked, because the second component has no rule for rewriting string \( b^iS_3 \).

- \( (S_1, S_2, S_3) \Rightarrow \text{rew} \ (a^iQ_2, b^iS_2, c^iS_3) \Rightarrow \text{comm} \ (a^iS_2, S_2, c^iS_3) \)

The derivation gets blocked, because the first component has no rule to rewrite string \( a^iS_2 \).

- \( (S_1, S_2, S_3) \Rightarrow \text{rew} \ (a^iQ_2, b^iQ_3, c^iS_3) \Rightarrow \text{comm} \ (a^iQ_2, b^icS_3, S_3) \Rightarrow \text{rew} \ (a^ib^ic^id, u, cS_3) \)

where \( u \in \{bS_2, bQ_3\} \)

This derivation sequence terminates with a terminal string of the form \( a^ib^iec^id \) for certain \( i \). Therefore the language generated by the returning PC grammar system \( \Gamma \) working by communication protocol (a) is \( L(\Gamma) = \{a^nb^iec^id | n \geq 1\} \).
Chapter 4

High-level replacement programs (HLRP)

4.1 Introduction

In this chapter definition of programmed high-level replacement systems will be given. That is, systems performing programmed rewriting on objects of some category will be defined. In the subsequent text such systems will be called high-level replacement programs. The term "high-level replacement program" will be abbreviated as HLRP. The definition of HLRPs is analogous to the graph transformation programs defined in [14], except that parallel composition of programs will be defined too. In fact HLRPs can be viewed as generalization of graph transformational programs defined in [14].

First, in Section 4.2 some extra conditions for categories which are treated in this chapter will be given. This will be needed in Section 4.4, where the definition of HLRPs will be given. Section 4.3 contains constructions needed to define the semantics of HLRPs.

4.2 HLRP Categories

The aim of this section is to give a description of categories which are suitable for defining high-level replacement programs over. Since to define HLRPs we need to be able to define high-level replacement productions and their parallel composition, these categories should satisfy HLR1 conditions. Besides, these categories should satisfy some other conditions in order to enable constructions which will be used for defining the semantics of high-level replacement programs. Conditions which categories are required to satisfy in order to make it possible to define HLRPs over them will be called HLRP conditions. Categories that satisfy HLRP conditions will be called HLRP categories.

Consider a category $CAT$ and a class of morphisms $M$. We require $CAT$
with class of morphisms $M$ to be a HLR1 category and to satisfy some extra conditions. However set like categories such as sets, graphs and hierarchical constructions do satisfy these extra conditions, so the restrictions introduced below will have little effect on the range of possible applications of the concepts described in this chapter. The conditions which are required are listed below.

**Definition 4.2.1.** A category $\text{CAT}$ together with class of arrows $M$ is said to satisfy **HLRP conditions** (in short, category $\text{CAT}$ with class of arrows $M$ is a HLRP category) if and only if it satisfies each of the following conditions:

1. (a) If $A \rightarrow B$ belongs to $M$ then the pushout of $A \rightarrow B$ and $A \rightarrow C$ exists. That is, the following diagram is a pushout.

   \[
   \begin{array}{c}
   A \\
   \downarrow \\
   C \\
   \downarrow \\
   D \\
   \end{array}
   \rightarrow \begin{array}{c}
   B \\
   \downarrow \\
   D \\
   \end{array}
   
   Besides, $A \rightarrow B$ belongs to $M$ implies that $C \rightarrow D$ belongs to $M$.

   (b) If $C \rightarrow D$ and $B \rightarrow D$ are in $M$, then the pullback of $C \rightarrow D$ and $B \rightarrow D$ exists. That is, the following diagram is a pullback.

   \[
   \begin{array}{c}
   A \\
   \downarrow \\
   C \\
   \downarrow \\
   D \\
   \end{array}
   \rightarrow \begin{array}{c}
   B \\
   \downarrow \\
   D \\
   \end{array}
   
   Besides, in this case $A \rightarrow B$ and $A \rightarrow C$ are in $M$ too.

2. Consider the pushout diagram

   \[
   \begin{array}{c}
   A \\
   \downarrow \\
   C \\
   \downarrow \\
   D \\
   \end{array}
   \rightarrow \begin{array}{c}
   B \\
   \end{array}
   
   If $A \rightarrow B$, $A \rightarrow C$ are in $M$, then the diagram above is also a pullback.

3. For all $A, B$ objects of $\text{CAT}$ their binary coproduct $A + B$ exists. Besides, for all arrows $f : A \rightarrow B$ and $g : A' \rightarrow B'$ such that $f$ is in $M$ and $g$ is in $M$ the morphism $f + g : A + A' \rightarrow B + B'$ is in $M$ too.
4. Consider the following diagram:

\[
\begin{array}{c c c c c c c c c}
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
(1) & (2) & (3)
\end{array}
\]

If \( A \to B, C \to E, C \to D, D \to F, E \to F \) are in \( M \), \((1) + (2)\) is a pushout and \((2)\) is a pullback, then \((1)\) and \((2)\) are pushouts.

5. Consider the following pushout diagram:

Assume that \( A \to B, A \to C, C \to D, B \to D \) are in \( M \). Consider an arbitrary arrow \( f : N \to D \). Let \( N_C, N_B \) and \( N_A \) be the pullback objects of \((C \to D \leftarrow N)\), \((B \to D \leftarrow N)\) and \((A \to B \to D \leftarrow N)\) respectively. That is, diagrams \((1), (2)\) and \((3)\) below are pullbacks.

\[
\begin{array}{c c c c c c c c c}
N_C & \to & C & \to & N & \to & B & \to & A \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
(1) & (2) & (3) \\
N & \to & D & \to & N & \to & D & \to & N \\
\end{array}
\]

By the pullback property of \( N_C \) and \( N_B \) there exist unique arrows \( N_A \to N_C \) and \( N_A \to N_B \) such that \( N_A \to N_C \to C = N_A \to A \to C \), \( N_A \to N_B \to B = N_A \to A \to B \) and \( N_A \to N_C \to N = N_A \to N_B \to N = N_A \to N \). Then the pushout of \( N_C \leftarrow N_A \to N_B \) is \( N \). That is, the diagram below is a pushout:

\[
\begin{array}{c c c c c c c c c}
N_A & \to & N_C & \to & N & \to & N_B & \to & N \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
(1) & (2) & (3) \\
N & \to & N & \to & N & \to & N \\
\end{array}
\]
Besides, the morphism \( u : N \rightarrow D \) induced by morphisms \( N_C \rightarrow C \rightarrow D \) and \( N_B \rightarrow B \rightarrow D \) coincides with the morphism \( f : N \rightarrow D \).

6. Consider the following diagram

\[
\begin{array}{c}
\text{K} \\
\downarrow \downarrow \downarrow \downarrow \\
\text{A} & \text{B} & \text{C} \\
\downarrow \downarrow \downarrow \downarrow \\
\text{E} & \text{D} & \text{F} \\
\downarrow \downarrow \downarrow \downarrow \\
\text{Z}
\end{array}
\]

Assume that (1),(2) are pushouts, (3),(4) are pullbacks, and all arrows of the diagram are in \( M \). Then the diagrams (5) and (6) are pushouts:

\[
\begin{array}{c}
\text{A} \leftarrow \text{K} \rightarrow \text{C} \\
\downarrow \downarrow \downarrow \\
\text{E} & \text{Z} & \text{F}
\end{array}
\]

7. (a) All arrows from class \( M \) are monomorphisms
(b) All isomorphisms and identity arrows are in \( M \)
(c) Class \( M \) is closed under composition of arrows
(d) If \( f : A \rightarrow B \) is in \( M \) and there exists such an arrow \( g : B \rightarrow A \) that \( f \circ g = id_B \) then \( f \) is an isomorphism
(e) Consider the following diagram:

\[
\begin{array}{c}
\text{C} \xrightarrow{f} \text{A} \\
\downarrow \downarrow \downarrow \\
\text{B} & \text{D} & \text{Z}
\end{array}
\]

\[
\begin{array}{c}
\text{C} \xrightarrow{f} \text{A} \\
\downarrow \downarrow \downarrow \downarrow \\
\text{B} & \text{D} & \text{Z}
\end{array}
\]
4.2. HLRP CATEGORIES

Assume that \( g, f, k, l \) are in \( M \), (1) is a pushout and (2) is a pullback. Then the unique morphism \( D \to Z \) induced by morphisms \( l \) and \( k \) are in \( M \) too.

8. Let be \( h : A \to B \) an arrow. Then there exists an object \( C \), called the kernel of \( h \), such that

\[
\begin{array}{ccc}
C & \xrightarrow{f} & A \\
\downarrow{g} & & \downarrow{h} \\
A & \xrightarrow{h} & B
\end{array}
\]

is a pullback, and there exists an object \( \tilde{A} \) such that \( \tilde{A} \) is the pushout of \( A \leftarrow C \to A \). That is, the diagram below is a pushout:

\[
\begin{array}{ccc}
C & \xrightarrow{f} & A \\
\downarrow{g} & & \downarrow{l'} \\
A & \xrightarrow{l} & \tilde{A}
\end{array}
\]

Consider the morphism \( u_h : \tilde{A} \to B \) induced by the arrow \( h \) according to the pushout property of \( \tilde{A} \). That is, \( u_h \) is the unique morphism such that \( h = u_h \circ l = u_h \circ l' \). We require \( u_h \) to belong to \( M \). Notice that because of HLPR condition 7a we get that \( l = l' \). The 3-tuple \( (\tilde{A}, u_h, l) \) is called the image of \( A \) under \( h \).

9. Let be \( r : A \to B \) in \( M \). If for some \( g : B \to D \) there exists an object \( C \) and arrows \( f : A \to C \) and \( h : C \to D \), such that the diagram

\[
\begin{array}{ccc}
A & \xrightarrow{r} & B \\
\downarrow{f} & & \downarrow{g} \\
C & \xrightarrow{h} & D
\end{array}
\]

is a pushout then such a \( C \) together with arrows \( f \) and \( h \) is unique up to isomorphism: if the diagram below is a pushout

\[
\begin{array}{ccc}
A & \xrightarrow{r \quad f'} & B \\
\downarrow{f} & & \downarrow{g} \\
C' & \xrightarrow{h'} & D
\end{array}
\]
then $C' \cong C$ and $f = \phi \circ f'$ and $h = h' \circ \phi^{-1}$ and $\phi : C' \to C$ is an isomorphism Recall, that such an object is called pushout-complement object. So, in other words: If $A \to B$ in $M$ and for a $B \to C$ there exists a pushout complement, then the pushout complement object is unique up to isomorphism.

Note that the first four conditions are in fact the classical HLR1 conditions described in Section 2.2. They are needed to enable us to define parallel composition of high-level replacement productions. HLRP conditions 7 and 8 are natural generalizations of properties of injective graph and set morphisms. HLRP condition 7 implies that objects of $\text{CAT}$ together with arrows belonging to $M$ form a subcategory of the category $\text{CAT}$.

HLRP conditions 5 and 6 are used to prove correctness of constructions presented in Subsection 4.3.2 and Subsection 4.3.3.

HLRP condition 8 enables us to reduce several problems to the case where all morphisms are taken from $M$.

HLRP condition 9 ensures that the result of applying a production rule to an object through a certain morphism is unique up to isomorphism.

Note, that categories of sets and graphs satisfy HLRP conditions. Besides, if the category $D$ satisfies HLRP conditions then categories of the form $N_{D,TG,A_g}$ (see Subsection 5.2.1) satisfy HLRP conditions too.

The following theorems will be useful for proving correctness of the constructions presented in this chapter.

**Theorem 4.2.1.** Let $\text{CAT}$ with class of arrows $M$ be a HLRP category. Consider the following diagram $\Delta$:

\[
\begin{array}{c}
G_0 \leftarrow D_0 \rightarrow G_1 \leftarrow D_1 \rightarrow G_2 \ldots G_{n-1} \leftarrow D_{n-1} \rightarrow G_n
\end{array}
\]

If all arrows $D_i \to G_i$ and $D_i \to G_{i+1}$ ($0 \leq i \leq n-1$) are in $M$ then there exists an object $PG$ and arrows $\pi_i : G_i \to PG$ and $\mu_j : D_j \to PG$ ($0 \leq j \leq n-1, 0 \leq i \leq n$) such that $PG$ together with arrows $\pi_i$ and $\mu_j$ is the colimit of $\Delta$. Besides, for each $i = 0 \ldots n$ and for each $j = 0 \ldots m-1$ arrows $\pi_i$ and $\mu_j$ are in $M$.

**Proof.** By induction on $n$. For $n = 1$ the colimit of $\Delta$ is the pushout of $G_0 \leftarrow D_0 \rightarrow G_1$. By HLRP condition 1a we get that the following diagram is a pushout, besides $\pi_i$ ($= 0,1$) are in $M$.

\[
\begin{array}{c}
D_0 \to \rightarrow G_1 \\
\downarrow \pi_1 \\
G_0 \to \rightarrow PG
\end{array}
\]

In this case $\mu_0$ is of the form $\mu_0 = D_0 \to G_0 \pi_0 \to PG$. By HLRP condition 7c we get that $\mu_0$ is in $M$ too.
4.2. **HLRP CATEGORIES**

Assume that for all $1 \leq k < n$ the statement of the theorem holds. Let $DG$ with arrows $p_i : G_i \to DG$ and $m_j : D_j \to PG$ ($0 \leq i \leq n-1$, $0 \leq j \leq n-2$) be the colimit of the following diagram:

$$G_0 \leftarrow D_0 \to G_1 \leftarrow D_1 \to G_2 \cdots G_{n-2} \leftarrow D_{n-2} \to G_{n-1}$$

Then by the induction hypothesis we get that such $DG$ and arrows $p_i : G_i \to DG$, $m_j : D_j \to Z$ indeed exist and for all $i = 0 \ldots n - 1$ and $j = 0 \ldots n - 2$ $p_i$ is in $M$ and $m_j$ is in $M$. Let $D$ with arrows $f : G_{n-1} \to D$ and $g : G_n \to D$ be the pushout of $G_{n-1} \leftarrow D_{n-1} \to G_n$. By HLRP condition 1a such an object $D$ and arrows $f$ and $g$ exist and $f$ and $g$ are in $M$. By HLRP condition 1a the pushout $D \leftarrow G_{n-1} \xrightarrow{p_i} DG$ exists. Let $PG$ with arrows $h : DG \to PG$ and $k : D \to PG$ be the pushout of $D \leftarrow G_{n-1} \xrightarrow{p_i} DG$. It is easy to see that $PG$ with arrows $\pi_i = h \circ p_i$, $\pi_n = k \circ g$ and $\mu_j = h \circ m_j$, $\mu_{n-1} = h \circ \pi_{n-1}$ ($0 \leq i \leq n - 1, 0 \leq j \leq n - 2$) is the colimit of $\Delta$. Since $M$ is closed under composition of arrows we get that $\pi_i, \mu_j$ are in $M$.

An analogous result can be proven for limits.

**Theorem 4.2.2.** Let CAT with class of arrows $M$ be a HLRP category. Consider the following diagram $\Delta$:

$$D_0 \to G_1 \leftarrow D_1 \to G_2 \cdots G_{n-1} \leftarrow D_{n-1} \to G_n \leftarrow D_n$$

If all arrows $G_i \leftarrow D_i$ and $D_{i-1} \to G_i$ ($1 \leq i \leq n$) are in $M$ then there exists an object $Z$ and arrows $\pi_i : Z \to G_i$ and $\mu_j : Z \to D_j$ ($0 \leq j \leq n, 1 \leq i \leq n$) such that $Z$ together with arrows $\pi_i$ and $\mu_j$ is the limit of $\Delta$. Besides, for each $i = 1 \ldots n$ and for each $j = 0 \ldots n$ arrows $\pi_i$ and $\mu_j$ are in $M$.

**Proof.** The proof goes by induction on $n$. For $n = 1$ the limit of the diagram $\Delta$ is the pullback of $D_0 \to G_1 \leftarrow D_1$. Since $D_0 \to G_1$ ($i = 0,1$) is in $M$, the pullback exists. Assume that $Z$ with arrows $\mu_i : Z \to D_i$ ($i = 0,1$) is the pullback of $D_0 \to G_1 \leftarrow D_1$. Let $\pi_0 = Z \xrightarrow{\mu_0} D_0 \to G_1$. By HLRP condition 1b $\mu_i$ ($i = 0,1$) is in $M$, therefore $\pi_0$ belongs to $M$ too. Assume that the statement of the theorem holds for all $1 \leq k < n$. Then consider the following diagram:

$$D_0 \to G_1 \leftarrow D_1 \cdots D_{n-2} \to G_{n-1} \leftarrow D_{n-1}$$

By the induction hypothesis there exists an object $Z'$ with arrows $p_i : Z' \to G_i$ and $m_j : Z' \to D_j$ ($1 \leq i \leq n-1, 0 \leq j \leq n-1$) such that $Z'$ with arrows $p_i$ and $m_j$ is the limit of the diagram above and arrows $p_i, m_j$ are in $M$ ($1 \leq i \leq n-1$, $0 \leq j \leq n-1$). Assume that $D$ with arrows $f : D \to D_{n-1}$ and $g : D \to D_n$ is the pullback of $D_{n-1} \to G_n \leftarrow D_n$. By HLRP condition 1b this pullback exists, besides $f$ and $g$ are in $M$. Then again the pullback of $D \xrightarrow{f} D_{n-1} \xleftarrow{p_i} Z'$ exists.
That is, the following diagram is a pullback

\[
\begin{array}{ccc}
Z & \xrightarrow{h} & Z' \\
\downarrow{k} & & \downarrow{p_{n-1}} \\
D & \xrightarrow{f} & D_{n-1}
\end{array}
\]

and \( k, h \) are in \( M \). It is easy to see that \( Z \) with arrows \( \pi_i = p_i \circ h, \mu_i = m_i \circ h \) and \( \pi_n = Z \xrightarrow{k} D \xrightarrow{g} D_n \rightarrow G_n \) and \( \mu_n = g \circ k (0 \leq j \leq n-1, 1 \leq i \leq n-1) \) is the limit of the diagram \( \Delta \). Since \( M \) is closed under composition of arrows we also get that \( \pi_i \) and \( \mu_j \) are in \( M \) \( (0 \leq j \leq n, 1 \leq i \leq n) \). \( \square \)

4.3 Derivation sequences in HLRP categories

This section still contains preparatory steps needed to define semantics of high-level replacement programs. Semantics of programs will be defined through sequences of direct derivations, so some basic facts and constructions should be introduced. This is done in this section. First some basic notations concerning sequences of direct derivations will be presented. Then some more involved constructions will be given together with the proof of their correctness. First, the notion of active derivation sequence derived from a derivation sequence will be given. The idea is roughly the following: if we have a derivation sequence with a starting object \( G \), then we are using only some parts of the object during the derivation. So it is possible to construct a derivation sequence by using the same rules but starting not from \( G \) but from its subobject which consists of exactly those elements of \( G \) which are used by productions of the derivation sequence. The second notion of this section is the notion of direct derivation equivalent to a derivation sequence. As the name reveals, it is nothing else, but a construction of a production and an occurrence morphism such that when it is applied to the starting object of a derivation sequence, it simulates the effect of that derivation sequence.

Consider an arbitrary category \( \text{CAT} \) with class of arrows of that category \( M \). Assume that \( \text{CAT} \) with class of arrows \( M \) satisfies HLRP conditions. Unless noted otherwise, all constructions presented in this section are thought to be defined implicitly over the category \( \text{CAT} \) and class of arrows \( M \).

4.3.1 Basic notations

Consider derivation sequences \( \tau_1 = G_1 \Rightarrow G_2 \ldots G_{n-1} \Rightarrow G_n \) and \( \tau_2 = H_1 \Rightarrow H_2 \ldots H_{m-1} \Rightarrow H_m \). Derivation sequences \( \tau_1 \) and \( \tau_2 \) are said to be \textit{composable} if \( H_1 \) coincides with \( G_n \). Composition of \( \tau_1 \) and \( \tau_2 \) is the derivation sequence \( \tau = \tau_1 \circ \tau_2 = G_1 \Rightarrow G_2 \ldots G_{n-1} \Rightarrow G_n \Rightarrow H_2 \cdot \cdot \cdot \Rightarrow H_m \).

If a derivation sequence \( \tau \) is of the form \( \tau = G_0 \Rightarrow G_1 \ldots G_{n-1} \Rightarrow G_n \) then the length of the derivation sequence \( \tau \) is \( n \) which is denoted by \( |\tau| \). Denote by
last(\tau) := G_n, the last element of the derivation sequence \tau and by first(\tau) := G_0 the first element of the derivation sequence \tau.

4.3.2 Construction of the ”active” derivation sequence

Let \mathcal{P} = \{p_i = (L \leftarrow K \rightarrow R) | i = 1 \ldots n\} be a class of rules and consider a derivation sequence:

\tau = G_0 \Rightarrow p_{1,o_1} G_1 \Rightarrow p_{2,o_2} G_2 \cdots \Rightarrow p_{n,o_n} G_n \quad (4.1)

Assume that for each \( i = 1 \ldots n \) the direct derivation \( G_{i-1} \Rightarrow p_{i,o_i} G_i \) from the derivation sequence \( \tau \) in (4.1) is of the form:

\[ p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i) \]

That is, diagrams (1) and (2) are pushouts. Now think of \( G_0 \) as a set or a graph. Some elements of \( G_0 \) might be not used by any of the productions \( p_i \) applied through \( o_i \) in the derivations sequence (4.1). That is, object \( G_0 \) might contain such an element \( e \), that \( e \) doesn’t belong to the image of the left-hand side of any of the productions \( p_i \). Formally: \( e \notin \bigcup^n_{i=1} o_i(L_i) \) where \( p_i = (L_i \leftarrow K_i \rightarrow R_i) \).

The aim of this subsection is to construct a derivation sequence \( ACT(\tau) \) of the same the length as \( \tau \) such that \( ACT(\tau) \) applies the productions \( p_i \) in the same order as \( \tau \) does and it uses almost the same occurrence morphisms but the first element of \( ACT(\tau) \) is different. The first element of the constructed derivation sequence should have the following property: all its elements or components must belong to the image of the left-hand side of a certain production \( p_i \). The subobject of \( G_0 \) with such a property will be called the initial object of the derivation sequence (4.1).

To be able to define the subobject described above, first we should define an object which is exactly made of the elements of the derivation sequence \( \tau \). Formally it looks like this:

**Definition 4.3.1.** Consider a derivation sequence \( \tau \) of the form (4.1). The colimit \( PG \) of the diagram

\[ G_0 \leftarrow D_0 \rightarrow G_1 \ldots G_i \leftarrow D_i \rightarrow G_{i+1} \ldots G_{n-1} \leftarrow D_{n-1} \rightarrow G_n \]

is called the underlying object of the derivation sequence \( \tau \)

In case of graphs or sets the underlying object simply the union of all elements of the graphs (sets) \( G_i \). Generally, the following statement holds for the underlying object of an arbitrary derivation sequence.
Claim 4.3.1. For an arbitrary derivation sequence $\tau$ of the form (4.1) the underlying object of $\tau$ defined in Definition 4.3.1 exists. Besides, for each $i = 0 \ldots n$ arrow $G_i \to PG$ belongs to $M$.

Proof. Notice that because of HLRP condition 1a arrows $D_i \to G_i$ and $D_i \to G_{i+1}$ are in $M$ ($0 \leq i \leq n-1$). Thus, from Theorem 4.2.1 it follows that the underlying object of an arbitrary derivation sequence is well defined. Besides for all $i = 1 \ldots n$ we have that $G_i \to PG$ belongs to $M$. \qed

We need to define the union of images of the left hand-sides. Union is defined usually through pushouts. The rather technical definition below is a category theoretical encoding of union of the images of several objects.

Definition 4.3.2. Consider the following diagram:

\[
\begin{array}{cccc}
A_1 & A_2 & \cdots & A_{n-1} & A_n \\
\alpha_1 & \alpha_2 & \cdots & \alpha_{n-1} & \alpha_n \\
& PG & & & \\
\end{array}
\]

Let $O = \{o_i | i = 1 \ldots n\}$. Then the union of images of objects $A_i$ under arrows $\alpha_i$ is the 3-tuple $\bigcup O = (\hat{A}, u: \hat{A} \to PG, \{f_i : A_i \to \hat{A} | 1 \leq i \leq n\})$ given by the following recursive definition:

- $n = 2$ Let $\hat{A}$ with arrows $h_i : \hat{A} \to A_i$ ($i = 1, 2$) be the pullback object of $A_1 \to PG \leftarrow A_2$. Then $\hat{A}$ is the pushout object of $A_1 \leftarrow \hat{A} \rightarrow A_2$. That is, the diagram (1) below is a pullback and the diagram (2) below is a pushout.

\[
\begin{array}{ccc}
\hat{A} & \xrightarrow{h_1} & A_1 \\
\downarrow h_2 & & \downarrow h_2 \\
A_2 & \xrightarrow{o_2} & PG \\
\end{array}
\quad \begin{array}{ccc}
\hat{A} & \xrightarrow{h_1} & A_1 \\
\downarrow f_2 & & \downarrow f_2 \\
A_2 & \xrightarrow{f_2} & A \\
\end{array}
\]

Let $u : \hat{A} \to PG$ be the unique arrow induced by arrows $o_1$ and $o_2$. That is, $u$ is the unique arrow such that $u \circ f_i = o_i$ ($i = 1, 2$). Then define the image of objects $A_i$ under arrows $\alpha_i$ to be $\bigcup O = (\hat{A}, u, \{f_i | i = 1, 2\})$.

- $n > 2$ Consider the class of arrows $O' = \{o_i | i = 1 \ldots n-1\}$. Construct the union of images $\bigcup O' = (\hat{A}', u : \hat{A}' \to PG, \{g_i | i = 1 \ldots n-1\})$ of objects $A_i$ under arrows $o_i$ for $i = 1 \ldots n-1$. Consider $\bigcup \{u, o_n\} = (\hat{A}, u : \hat{A} \to PG, \{g : \hat{A} \to A, f_n : A_n \to \hat{A}\})$ - the union of images of $\hat{A}'$ under $u'$ and $A_n$ under $o_n$. Then define the union of images of objects $A_i$ under arrows $o_i$ ($1 \leq i \leq n$) to be the following 3-tuple $\bigcup O = (\hat{A}, u, \{f_i | i = 1 \ldots n\})$ where $f_i = g \circ g_i$ for each $i = 1 \ldots n-1$. 

4.3. DERIVATION SEQUENCES IN HLRP CATEGORIES

For sets and graphs \( \vec{A} \) is the union of images of \( A_i \)'s under morphisms: \( \alpha_i : A_i \rightarrow PG \). That is, \( \vec{A} = \bigcup_i^{n} \alpha_i(A_i) \).

The proof of correctness of the construction is given below.

**Claim 4.3.2.** Assume that \( \alpha_i : A_i \rightarrow PG \) are in \( M \) for \( 1 \leq i \leq n \). Then the union of images \( \bigcup \{ \alpha_i | 1 \leq i \leq n\} = (\vec{A}, u : \vec{A} \rightarrow PG, \{f_i|1 \leq i \leq n\}) \) of objects \( A_i \) under arrows \( \alpha_i \) is well defined. Besides arrow \( u \) belongs to \( M \) and \( u \circ f_i = \alpha_i \) for each \( i = 1 \ldots n \).

**Proof.** The proof goes by induction on \( n \). For \( n = 2 \), the pullback of \( A_1 \rightarrow PG \) and \( A_2 \rightarrow PG \) exists, that is, the object \( \vec{A} \) and the arrows \( h_1 \) and \( h_2 \) are well defined. Then the pushout of \( A_1 \xleftarrow{g_1} \vec{A} \xrightarrow{g_2} A_2 \) exists. That is, object \( \vec{A} \) is well defined. Besides, since \( \alpha_i : A_i \rightarrow PG \) are in \( M (i = 1, 2) \), and \( \vec{A} \) is the pullback of \( A_1 \xrightarrow{\alpha_1} PG \xleftarrow{\alpha_2} A_2 \) by the HLRP condition 7e the induced arrow \( u : \vec{A} \rightarrow PG \) is in \( M \) too. From the definition of \( u \) we get that \( u \circ f_i = \alpha_i \) \( (i = 1, 2) \) indeed holds.

Assume that the union \( \bigcup \{ \alpha_i | 1 \leq i \leq n - 1\} = (\vec{A}', u : \vec{A}' \rightarrow PG, \{g_i|1 \leq i \leq n - 1\}) \) of images of objects \( A_i \) under arrows \( \alpha_i \) \( (1 \leq i \leq n - 1) \) is well defined, besides \( u \) is in \( M \) and \( u \circ g_i = \alpha_i \) for all \( 1 \leq i \leq n - 1 \).

We need to prove that the union \( \bigcup \{ \alpha_i | 1 \leq i \leq n\} = (\vec{A}, u, \{f_i|1 \leq i \leq n\}) \) of images of objects \( A_i \) under arrows \( \alpha_i \) \( (1 \leq i \leq n) \) is well defined, \( u \) is in \( M \) and \( u \circ f_i = \alpha_i \) for all \( 1 \leq i \leq n \). From the case for \( n = 2 \) we get that \( \bigcup \{ \alpha_i, u_i \} = (\vec{A}, u, \{g : \vec{A}' \rightarrow \vec{A}, f_i : A_n \rightarrow \vec{A}\}) \) is well defined, \( u \) is in \( M \) and \( u \circ g = u \) and \( u \circ f_n = \alpha_n \). Besides by taking \( g \circ g_i \) for \( f_i \) \( (1 \leq i \leq n - 1) \) we get that \( u \circ f_i = u \circ g \circ g_i = u' \circ g_i = \alpha_i \) for \( 1 \leq i \leq n - 1 \) and \( u \circ f_n = \alpha_n \). \( \square \)

Now we are ready to define the initial object of a derivation sequence in terms of category theory. It will be the intersection (pullback) of union of the images of the left-hand sides with the starting (first) object \( G_0 \) of the derivation sequence (4.1). Here we explore the following HLRP property: for each arrow \( A \rightarrow B \) there exists the image object \( \vec{A} \) of \( A \) under this arrow. This allows to take the union of the images of the left-hand sides, where union is meant to be the construction introduced in the previous definition.

**Definition 4.3.3.** Consider a derivation sequence \( \tau \) of the form (4.1). For each \( i = 1 \ldots n \) let \( (L_i', u_{i}, g_i) \) be the image of \( L_i \) under \( \alpha_i \). Let \( \bigcup \{L'_i \xrightarrow{u_{i}} G_i \rightarrow PG|1 \leq i \leq n\} = (L', u_L, \{f_i|1 \leq i \leq n\}) \) be the union of images of objects \( L'_i \) under arrows \( L'_i \xrightarrow{u_{i}} G_i \rightarrow PG \). Then the initial object of the derivation sequence \( \tau \) is the object \( H_0 \) such that the diagram below is a pullback.

\[
\begin{array}{ccc}
H_0 & \longrightarrow & L' \\
\downarrow & & \downarrow \\
G_0 & \longrightarrow & PG
\end{array}
\]
The initial object of a derivation sequence is well defined. That is, the following holds:

**Claim 4.3.3.** In Definition 4.3.3 the initial object is well defined.

**Proof.** Since arrows \( u_{i+1} : L'_i \rightarrow G_i \) are in \( M \) \((1 \leq i \leq n)\) and \( G_i \rightarrow PG \) are in \( M \) then \( L'_i \overset{u_{i+1}}{\rightarrow} G_i \rightarrow PG \) are in \( M \) for all \( 0 \leq i \leq n - 1 \). By Claim 4.3.2 it implies that \( \tilde{L}' \) is well defined. Besides, \( u_L : \tilde{L}' \rightarrow PG \) is in \( M \). Then the pullback of \( \tilde{L}' \overset{u_L}{\rightarrow} PG \) and \( G_0 \rightarrow PG \) exists. That is, \( H_0 \) is well defined. \( \square \)

Now we are ready to present the main definition of this subsection:

**Definition 4.3.4.** Consider a derivation sequence \( \tau \) of the form (4.1). The **active derivation sequence** of the derivation sequence \( \tau \) (denoted by \( ACT(\tau) \)) is the derivation sequence

\[
H_0 \Longrightarrow p_1, H_1 \ldots H_{n-1} \Longrightarrow p_n, H_n
\]

such that for each \( i = 1 \ldots n \) the diagrams (1), (2), (3) and (4) below are pushouts

\[
\begin{array}{ccc}
L_i & \xleftarrow{D_i} & K_i \\
\downarrow & & \downarrow \\
H_{i-1} & \xrightarrow{D_{i-1}} & H_i \\
\downarrow & & \downarrow \\
G_{i-1} & \xleftarrow{D_{i-1}} & G_i \\
\end{array}
\]

and \( H_0 \) is the initial object of the derivation \( \tau \) and \( L_i \rightarrow H_{i-1} \rightarrow G_{i-1} = L_i \overset{p_i}{\rightarrow} G_{i-1} \).

For sets or graphs \( ACT(\tau) \) is nothing else but the derivation with the "same" occurrence morphisms starting not from the original set (graph), but a subset (subgraph) of that with the following property: it contains all elements of the original initial set (graph) which are used during the derivation. That is, all those elements of the initial set (graph) which do not occur in the image of the left-hand side of any of the rules \( p_i \) are omitted from the graphs constituting the derivation.

The fact that this construction can be done is not so obvious. Theorem 4.3.1 states that the notion of \( ACT(\tau) \) is well defined. The theorem is proven by constructing \( ACT(\tau) \) by induction on the length of the derivation sequence \( \tau \). Before turning to the theorem some elementary properties of \( ACT(\tau) \) will be stated.
4.3. DERIVATION SEQUENCES IN HLRP CATEGORIES

Lemma 4.3.1 (Elementary properties of the active derivations sequence).
Let \( \tau \) be a derivation sequence of the shape (4.1). Assume that \( ACT(\tau) \) exists.
Then using notations of Definition 4.3.4 the following properties hold: for each \( i = 0 \ldots n - 1 \).
(a) \( D'_i \to D_i, H_i \to G_i, D_i \to G_i, D'_i \to H_i \) are in \( M \) (\( l = i, i + 1 \)).
(b) Diagrams (3) and (4) are pullbacks.
(c) \( ACT(\tau) \) is unique (up to isomorphism)

Proof. Proof of part (a) By induction on \( i \). For \( i = 0 \) the statement holds,
since \( H_0 \to G_0 \) is in \( M \) because of inheritance of \( M \) arrows under pullback
(HLRP condition 1b). The rest of the part (a) follows from HLRP condition 1a. If for some \( i \) (a) holds then again because of HLRP condition 1a (a) holds for \( i + 1 \).

Proof of part (b) Because of HLRP condition 2 diagrams (3) and (4) are also pullbacks.

Proof of part (c) \( H_0 \) is unique, since it is the pullback of \( (G_0 \to PG \xleftarrow{\ell} \tilde{L}') \).
If for some \( i = 0 \ldots n - 1 \) object \( H_i \) is unique then \( D_i \) and \( H_{i+1} \) are
unique as well. Indeed, the object \( H_{i+1} \) is the result of applying the rule
\( p_{i+1} = (L_{i+1} \leftarrow K_{i+1} \to R_{i+1}) \) to \( H_i \) through \( L_{i+1} \to H_i \) and \( D'_i \) is the
pushout complement object of the derivation \( H_i \Rightarrow p_{i+1}, L_{i+1} \to H_i, H_{i+1} \).
From HLRP condition 9 it follows that the pushout complement object,
and therefore the result of the direct derivation \( H_i \Rightarrow p_{i+1}, L_{i+1} \to H_i, H_{i+1} \)
is unique (up to isomorphism). Thus, we get that \( ACT(\tau) \) is unique up to
isomorphism.

\[ \square \]

Theorem 4.3.1. For each \( \tau \) of the form (4.1) \( ACT(\tau) \) is well defined.

Proof. Assume that for each \( 0 \leq i \leq n \) \( \tau_i = \tau_i \circ \bar{\tau}_i \) and \( |\tau_i| = i \). We show how
to construct \( ACT(\tau_{k+1}) \) from \( ACT(\tau_k) \). The following claim will be needed:

Claim 4.3.4. Assume that it is possible to construct \( ACT(\tau_i) \) for each \( i = 0 \ldots k \). Then the following holds:
\[
\begin{align*}
G'_i & \text{ is the pullback of } G_i \to PG \xleftarrow{\ell} \tilde{L}' \\
\text{if and only if} & \\
G'_i & \text{ is the pullback of } H_i \to G_i \to PG \xleftarrow{\ell} \tilde{L}'
\end{align*}
\]
(4.2)

The proof of the claim will be given later. Consider the direct derivation
\( G_k \Rightarrow p_{k+1} \circ h_{k+1} \) \( G_{k+1} \). We know that \( L_{k+1} \xrightarrow{g_{k+1}} L'_{k+1} \xrightarrow{u_{k+1}} \tilde{L}' \xrightarrow{u} PG = L_{k+1} \xrightarrow{g_{k+1}} G_k \to PG \). Then from Claim 4.3.4 it follows that there exists a
unique \( u : L_{k+1} \to G''_k \) such that \( L_{k+1} \xrightarrow{u} G''_k \to H_k \to G_k = L_{k+1} \to G_k \). Let
$D'_k$ be the pullback of $H_k \rightarrow G_k \leftarrow D_k$. Let $H_{k+1}$ be the pushout object of $D'_{k+1} \leftarrow K_{k+1} \rightarrow R_{k+1}$. Then we have the following diagram:

Since (1) + (3) is a pushout, (3) is a pullback by HLRP property 3 we get that (3) is a pushout too. (2) and (4) are pushouts by construction. Note that $G_{k+1}$ is the pushout of $D_k \leftarrow D'_k \rightarrow H_{k+1}$ because of uniqueness of the pushout object. Thus, we have that $\omega = H_k \Rightarrow p_{k+1} H_{k+1}$. It is easy to see that $ACT(\tau_{k+1}) = ACT(\tau_k) \circ \omega$. \hfill \Box

Now the proof of Claim 4.3.4 will be given.

**Proof of Claim 4.3.4.** The following two statements are true:

**Claim 4.3.5.** If (4.2) holds for some $G_i$, then

$D'_i$ is the pullback of $D'_i \rightarrow D_i \rightarrow PG \xrightarrow{\nu} \hat{L}'$ 
if and only if $D'_i$ is the pullback of $D_i \rightarrow PG \xrightarrow{\nu} \hat{L}'$ \hspace{1cm} (4.3)

**Claim 4.3.6.** If (4.3) holds for certain $i$, then (4.2) holds for $i + 1$.

The claim follows from the two statements above and the definition of the initial object of a derivation sequence. \hfill \Box

**Proof of Claim 4.3.5.** Take the pullback object $D''_i$ of $D'_i \rightarrow H_i$ and $G''_i \rightarrow H_i$. Then:
• (i) The diagram (1) + (2) below is a pullback.

\[
\begin{array}{c}
D_i'' \rightarrow G_i'' \rightarrow \tilde{L}' \\
\downarrow \quad \downarrow \quad \downarrow \quad u_L \\
D_i' \rightarrow H_i \rightarrow G_i \rightarrow PG
\end{array}
\]

Indeed, (1) is a pullback and (2) is a pullback, which implies that (1) + (2) is also a pullback.

• (ii) The diagram ((1) + (3)) + (2) below is a pullback.

\[
\begin{array}{c}
D_i'' \rightarrow G_i'' \rightarrow \tilde{L}' \\
\downarrow \quad \downarrow \quad \downarrow \quad u_L \\
D_i' \rightarrow H_i \rightarrow G_i \rightarrow PG
\end{array}
\]

Indeed, (3) is a pushout with all arrows in \( M \), from which follows that (3) is a pullback. (1) is a pullback, which implies that (1) + (3) is a pullback. (2) is a pullback, from which it follows that (1) + (3) + (2) is a pullback.

So \( D_i' \) is the pullback object of \( D_i \rightarrow PG \leftarrow \tilde{L}' \) and \( D_i' \rightarrow PG \leftarrow \tilde{L}' \).

\[
\square
\]

**Proof of Claim 4.3.6.** Assume that \( G_{i+1}'' \) is the pullback of \( H_{i+1} \rightarrow G_{i+1} \rightarrow PG \not\subseteq \tilde{L}' \). Since \( H_{i+1} \rightarrow G_{i+1} \), \( G_{i+1} \rightarrow PG \) and \( u_L \) are in \( M \), the pullback of the diagram indeed exists. We will prove that \( G_{i+1}'' \) with arrows \( f : G_{i+1}'' \rightarrow H_{i+1} \rightarrow G_{i+1} \) and \( g : G_{i+1}'' \rightarrow \tilde{L}' \) is the pullback of \( G_{i+1} \rightarrow PG \not\subseteq \tilde{L}' \). It is clear that \( G_{i+1}'' \) with arrows \( f \) and \( g \) is a cone of the diagram \( G_{i+1} \rightarrow PG \not\subseteq \tilde{L}' \).

Consider now an arbitrary object \( N \) and morphisms \( N \rightarrow G_{i+1} \) and \( N \rightarrow \tilde{L}' \) such that

\[
N \rightarrow \tilde{L}' \not\subseteq PG = N \rightarrow G_{i+1} \rightarrow PG
\]

we will show that there exists an arrow \( N \rightarrow H_{i+1} \) such that \( N \rightarrow G_{i+1} = N \rightarrow H_{i+1} \rightarrow G_{i+1} \). Let \( N_i \) be the pullback of \( D_i' \rightarrow \tilde{L}' \leftarrow N \). Then \( N_i \) is the pullback of \( D_i \rightarrow PG \leftarrow N \) and \( D_i' \rightarrow PG \leftarrow N \). It can be illustrated by the
following diagram:

\[
\begin{array}{cccc}
N_1 & \rightarrow & D'_i & \rightarrow & D'_i & \rightarrow & D_i \\
\downarrow (1) & & \downarrow (2) & & \downarrow & & \downarrow \\
N & \rightarrow & \tilde{L}' & \rightarrow & \tilde{L}' & \rightarrow & PG
\end{array}
\]

Here (1) and (2) are pullbacks, which implies that (1) + (2) is also a pullback. But \( D'_i \) is the pullback of \( D'_i \rightarrow PG \leftarrow \tilde{L}' \). That is:

\[
\begin{array}{cccc}
N_1 & \rightarrow & D'_i & \rightarrow & D'_i & \rightarrow & D_i \\
\downarrow (1) & & \downarrow (3) & & \downarrow & & \downarrow \\
N & \rightarrow & \tilde{L}' & \rightarrow & \tilde{L}' & \rightarrow & PG
\end{array}
\]

where (1) and (3) are pullbacks, from which it follows that (1) + (3) is a pullback too.

From this and (4.4) it follows that \( N_1 \) is the pullback of \( N \rightarrow G_{i+1} \rightarrow PG \) and \( D_i \rightarrow G_{i+1} \rightarrow PG \). Since \( G_{i+1} \rightarrow PG \) is in \( M \) we have that \( G_{i+1} \rightarrow PG \) is a monomorphism which implies that \( N_1 \) is the pullback of \( N \rightarrow G_{i+1} \) and \( D_i \rightarrow G_{i+1} \).

By a similar argument we get that \( N_1 \) is the pullback of \( D'_i \rightarrow H_{i+1} \rightarrow G_{i+1} \) and \( N \rightarrow G_{i+1} \). Let \( N_3 \) be the pullback object of \( H_{i+1} \rightarrow G_{i+1} \leftarrow N \). That is, the following three diagrams are pullbacks:

\[
\begin{array}{cccc}
N_1 & \rightarrow & D'_i & \rightarrow & D'_i & \rightarrow & D_i \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
N & \rightarrow & G_{i+1} & \rightarrow & G_{i+1} & \rightarrow & G_{i+1}
\end{array}
\]

\[
\begin{array}{cccc}
N_1 & \rightarrow & D_i & \rightarrow & D_i & \rightarrow & H_{i+1} \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
N & \rightarrow & G_{i+1} & \rightarrow & G_{i+1} & \rightarrow & G_{i+1}
\end{array}
\]

Recall that the following diagram is a pushout:

\[
\begin{array}{cccc}
D'_i & \rightarrow & H_{i+1} & \\
\downarrow & & \downarrow & \\
D_i & \rightarrow & G_{i+1}
\end{array}
\]
From HLRP condition 5 we get that the diagram

\[
\begin{array}{ccc}
N_1 & \rightarrow & N_3 \\
\downarrow^{id_{N_1}} & & \\
N_1 & \rightarrow & N
\end{array}
\]

is a pushout. But \(N_3\) is the pushout of \(N_1 \xrightarrow{id_{N_1}} N_1\) and \(N_1 \rightarrow N_1\). Because of uniqueness of pushouts we get that \(N_3 \cong N\). Let \(iso : N \rightarrow N_3\) be the corresponding isomorphism. But

\[
N_3 \rightarrow H_{i+1} \rightarrow G_{i+1} = N_3 \xrightarrow{iso} N \rightarrow G_{i+1}
\]

Thus, we get that the arrow \(N \rightarrow H_{i+1}\) satisfies the following equality:

\[
N \rightarrow H_{i+1} \rightarrow G_{i+1} = N \rightarrow G_{i+1}
\]

where \(N \rightarrow H_{i+1} = N \xrightarrow{iso} N_3 \rightarrow H_{i+1}\) It implies that

\[
N \rightarrow H_{i+1} \rightarrow G_{i+1} \rightarrow PG = N \rightarrow \tilde{L}' \xrightarrow{u} PG
\]

That is, \(N\) with arrows \(N \rightarrow H_{i+1}\) and \(N \rightarrow \tilde{L}'\) is a cone of the diagram \(H_{i+1} \rightarrow PG \xrightarrow{u} \tilde{L}'\).

Recall that \(G''_{i+1}\) is the pullback of \(H_{i+1} \rightarrow PG \xleftarrow{u} \tilde{L}'\). Now, because of the pullback property we get that there exists a unique \(u : N \rightarrow G''_{i+1}\) such that

\[
N \xrightarrow{u} G''_{i+1} \rightarrow H_{i+1} = N \rightarrow H_{i+1} \tag{4.5}
\]

\[
N \xrightarrow{u} G''_{i+1} \rightarrow \tilde{L}' = N \rightarrow \tilde{L}' \tag{4.6}
\]

From (4.5) we get that

\[
N \xrightarrow{u} G''_{i+1} \rightarrow H_{i+1} \rightarrow G_{i+1} = N \rightarrow H_{i+1} \rightarrow G_{i+1} \tag{4.7}
\]

Assume that there exists an arrow \(\tilde{u} : N \rightarrow G''_{i+1}\) satisfying (4.7) and (4.6). The arrow \(H_{i+1} \rightarrow G_{i+1}\) is in \(M\), which implies that \(H_{i+1} \rightarrow G_{i+1}\) is a monomorphism. That is, (4.7) implies that

\[
N \xrightarrow{\tilde{u}} G''_{i+1} \rightarrow H_{i+1} = N \rightarrow H_{i+1}
\]

In other words, \(\tilde{u}\) satisfies (4.5). But because of the pullback property of \(G''_{i+1}\) it implies that \(\tilde{u} = u\).

We have just proven that for all arrows \(N \rightarrow G_{i+1}\) and \(N \rightarrow \tilde{L}'\) satisfying (4.4) there exists a unique arrow \(u : N \rightarrow G''_{i+1}\) satisfying (4.6) and (4.7). This means that \(G''_{i+1}\) is the pullback of \(u_L : \tilde{L}' \rightarrow PG\) and \(\tilde{G}_{i+1} \rightarrow PG\). \(\square\)
4.3.3 Direct derivation equivalent to a derivation sequence

Consider again the derivation sequence (4.1). Again think of $G_0$ as a set or a graph. We would like to construct a production and an occurrence morphism with the property that the application of the production through the occurrence morphism simulates the effect of the derivation sequence $\tau$. That is, the application of the production via the occurrence morphism deletes exactly those elements of $G_0$ which are deleted by $\tau$ and adds exactly those elements, which are added by $\tau$.

To do this, we need to construct a production such that the left-hand side of the production would contain all elements of $G_0$ which are used at least by one derivation step of the derivation sequence. We need the gluing object of the derivation to contain all elements of $G_0$ which are not deleted by any derivation step of $\tau$ but are used at least by one derivation step, that is, occur in the intersection of the union of images of the left-hand sides with $G_0$. Besides, we need the right-hand side to contain all elements which have been eventually added to $G_0$ by $\tau$. That is, we want the left hand-side to be of the form $(\bigcup_{i=1}^{n} o_i(L_i)) \cap G_0$ we want the gluing object to be of the form $\bigcap_{i=1}^{n} o_i(l_i(R_i))$ and we want the right-hand side to be $(\bigcup_{i=1}^{n} h_i(R_i)) \cap G_n$.

Notice now that the initial object of $\tau$ meets our requirement for the left-hand side of the production. The last element of $ACT(\tau)$ has the property required for the right-hand side of the desired production. The intersection of $D_k$'s from $ACT(\tau)$ fulfills the criteria for the gluing object of the production. The occurrence morphism might be simply the morphism $H_0 \rightarrow G_0$. It is easy to see that the production chosen in a way sketched above fulfills our requirements in the case of graphs or sets. However, the category theoretical formulation of the construction is not so obvious.

**Definition 4.3.5.** Let $\tau$ be a derivation sequence of the form (4.1). Consider $ACT(\tau)$ defined in Definition 4.3.4:

$$
\begin{array}{cccccc}
H_0 & \leftarrow & D'_0 & \rightarrow & H_1 & \cdots & H_{n-1} & \leftarrow & D'_{n-1} & \rightarrow & H_n \\
| & & \downarrow & & \downarrow & & \cdots & & \downarrow & & \downarrow \\
G_0 & \leftarrow & D_0 & \rightarrow & G_1 & \cdots & G_{n-1} & \leftarrow & D_{n-1} & \rightarrow & D_n \\
\end{array}
$$

The direct derivation equivalent to $\tau$ (denoted by $PROD(\tau)$) is the following direct derivation

$$
\begin{array}{cccc}
H_0 & \leftarrow & K & \rightarrow & H_n \\
| & & \downarrow & & \downarrow \\
G_0 & \leftarrow & Z & \rightarrow & G_n \\
\end{array}
$$
where $K$ is the limit object of the diagram
\[
D'_0 \to H_1 \leftarrow D'_1 \ldots D'_i \to H_{i+1} \leftarrow D'_{i+1} \ldots H_{n-1} \leftarrow D'_n
\]
and $Z$ is the limit object of the diagram
\[
D_0 \to G_1 \leftarrow D_1 \ldots D_i \to G_{i+1} \leftarrow D_{i+1} \ldots G_{n-1} \leftarrow D_n
\]
and $K \to Z$ is the unique arrow induced by arrows $K \to D'_i \to D_i$ according to the limit property of $Z$. That is, $K \to H_i \to G_i = K \to Z \to G_i$ and $K \to H_j \to D'_j \to D_j = K \to Z \to G_j \to D_j \ (0 \leq i \leq n, 0 \leq j \leq n - 1)$.

To prove that the construction of Definition 4.3.5 is well defined HLRP condition 6 will be used. The following theorem states that the construction is correct.

**Theorem 4.3.2.** For all derivation sequences $\tau$ of the shape (4.1) $PROD(\tau)$ is well defined.

**Proof.** We need to prove that diagrams (I) and (II) below are indeed pushouts.

\[
\begin{array}{c}
\text{(I)} \\
H_0 \\ K \\ G_0 \\
\end{array}
\quad
\quad
\begin{array}{c}
\text{(II)} \\
H_n \\ Z \\ G_n \\
\end{array}
\]

Let $K_k$ be the limit object of the diagram
\[
D'_0 \to H_1 \leftarrow D'_1 \ldots D'_i \to H_{i+1} \leftarrow D'_{i+1} \ldots H_{k-1} \leftarrow D'_k
\]
Let $Z_k$ be the limit object of the diagram:
\[
D_0 \to G_1 \leftarrow D_1 \ldots D_i \to G_{i+1} \leftarrow D_{i+1} \ldots G_{k-1} \leftarrow D_k
\]
Note that $K_0 \cong D'_0$ and $Z_0 \cong D_0$. For some $1 \leq k < n$ let $K'_k$ be the pullback of $D'_k \to H_{k+1} \leftarrow D'_{k+1}$ and let $Z'_k$ be the pullback of $D_k \to G_{k+1} \leftarrow D_{k+1}$. Then by the proof of the Theorem 4.2.2 we have that diagrams (1) and (2) below are pullbacks.

\[
\begin{array}{c}
K_{k+1} \\ K'_k \\
\end{array}
\quad
\quad
\begin{array}{c}
Z_{k+1} \\ Z'_k \\
\end{array}
\]

Consider now the following diagram:

\[
\begin{array}{c}
H_0 \\ D'_0 \\ K_j \\
\end{array}
\quad
\quad
\begin{array}{c}
D'_j \\ H_j \\
\end{array}
\]

\[
\begin{array}{c}
G_0 \\ D_0 \\ Z_j \\
\end{array}
\quad
\quad
\begin{array}{c}
D_j \\ G_j \\
\end{array}
\]

\[
\begin{array}{c}
K \\ D' \\
\end{array}
\quad
\quad
\begin{array}{c}
Z \\
\end{array}
\]

\[
\begin{array}{c}
D'_0 \\
\end{array}
\quad
\quad
\begin{array}{c}
D'_j \\
\end{array}
\]

\[
\begin{array}{c}
H_j \\
\end{array}
\quad
\quad
\begin{array}{c}
H_j \\
\end{array}
\]

\[
\begin{array}{c}
K_k \\
\end{array}
\quad
\quad
\begin{array}{c}
Z_k \\
\end{array}
\]

\[
\begin{array}{c}
D_k \\
\end{array}
\quad
\quad
\begin{array}{c}
D_k \\
\end{array}
\]

\[
\begin{array}{c}
D' \\
\end{array}
\quad
\quad
\begin{array}{c}
H \\
\end{array}
\]

\[
\begin{array}{c}
D \\
\end{array}
\quad
\quad
\begin{array}{c}
G \\
\end{array}
\]

\[
\begin{array}{c}
G \\
\end{array}
\quad
\quad
\begin{array}{c}
G \\
\end{array}
\]

\[
\begin{array}{c}
D \\
\end{array}
\quad
\quad
\begin{array}{c}
D \\
\end{array}
\]

\[
\begin{array}{c}
H \\
\end{array}
\quad
\quad
\begin{array}{c}
H \\
\end{array}
\]
We will prove that for each \( j = 0 \ldots n \) diagrams (3) and (4) are pushouts. Since \((0)\) and \((0')\) are pushouts we get that \((0) + (3)\) and \((0') + (4)\) are pushouts. Note that for \( j = n \) we get that \( K_j \cong K \) and \( Z_j \cong Z \) and \((0) + (3)\) coincides with \((I)\) and \((0') + (4)\) coincides with \((II)\). Thus, if for all \( j = 0 \ldots n \) diagrams (3) and (4) are pushouts then we get that the diagrams \((I)\) and \((II)\) are pushouts. That is, the theorem is proven.

For \( k = 0 \) diagram (3) and (4) are indeed pushouts. Assume that for all \( j \leq k \) diagrams (3) and (4) are pushouts. Consider now the following diagram

\[
\begin{array}{c}
\text{By the induction assumption (5) and (6) are pushouts. By the HLRP condition 6 diagrams (8) and (7) are pushouts too. Using the fact that diagrams (1) and (2) are pullbacks and the HLRP condition 6 we get that diagrams (9) and (10) below are pushouts.}
\end{array}
\]

\[
\begin{array}{c}
\text{The fact that (5) and (9) are pushouts implies that (5) + (9) is a pushout. Similarly we get that (10) + (8) is a pushout too.}
\end{array}
\]

\section{High-Level Replacement Programs (HLRP)}

In this section high-level replacement programs (in short HLRPs), their syntax and semantics will be defined. One may think of a high-level replacement program as a programmed rewriting system analogous to programmed string
4.4. \textit{HIGH-LEVEL REPLACEMENT PROGRAMS (HLRP)}

grammars. But unlike most of the programmed rewriting systems defined in
the literature, high-level replacement programs allow programmed rewriting of
objects of an arbitrary category of a certain class.

Unlike programmed string grammars, HLRPs are given by an expression
rather than by an automaton. In fact, the definition of HLRP introduced here
can be viewed as generalization of the notion of graph transformational pro-
grams defined in [14]. In [14] a construction very similar to HLRPs was intro-
duced, but it allowed only rewriting of graphs and contained only sequential
control structures. On the contrary, high-level replacement programs encode
programmed applications of high-level replacement productions defined over an
arbitrary category. The only requirement set upon the category is that it has to
satisfy HLRP conditions. Besides, high-level replacement programs give oppor-
tunity to apply high-level replacement productions in parallel. For the category
of labeled graphs the syntax of HLRPs without parallel composition and non-
deterministic composition coincides with the syntax of graph transformational
programs. In this case the semantics of a HLRP viewed as a binary relation
coincides with that of the corresponding graph transformational program.

Subsection 4.4.1 describes the syntax and the intuitive semantics of pro-
grams. In Subsection 4.4.2 the formal semantics of programs will be given.

4.4.1 \textbf{Syntax of high-level replacement programs}

Consider a HLRP category $\textit{CAT}$ with class of arrows $M$. Let $\mathcal{P}$ be a set of
productions over $\textit{CAT}$. The syntax of high-level replacement programs defined
over $\textit{CAT}$ can be given in the following way.

\textbf{Definition 4.4.1 ( Syntax of high-level replacement programs )}. Consider
a category $\textit{CAT}$ with some class of arrows $M$. Let $\mathcal{P}$ be a set of high-level
replacement productions over $\textit{CAT}$ with class of arrows $M$. $\mathcal{P}$ is a \textit{well-formed
high-level replacement program} over $\textit{CAT}$ if and only if it arises by applying one
of the following rules:

- $P = \mathcal{P}' \subseteq \mathcal{P}$, provided that $\mathcal{P}'$ is finite, is a well-formed program. Such
  programs are called elementary programs.

- If $P_1$ and $P_2$ are well-formed programs, so is their sequential composition
  $P_1; P_2$.

- If $P$ is a well-formed program, so is its iteration $P \downarrow$.

- If $P_1, \ldots, P_n$ are well-formed programs, so is their parallel composition
  $||^nP_i$

- If $P_1, \ldots, P_n$ are well-formed programs then so is their weak parallel com-
  position $||^nP_i$

- If $P_1, \ldots, P_n$ are well-formed programs then so is their non-deterministic
  composition $\bigodot^nP_i$
The intuition behind the definition is the following. If an elementary program is applied to an object then the resulting object arises as the result of application of one of the productions constituting the program. If none of the productions of the program can be applied to the given object, then the effect of applying the program to the object is undefined. If a program is of the form \( P = P_1 ; P_2 \), then the result of applying the program \( P \) to an object \( G \) is the same as applying \( P_2 \) to an object \( H \) which has been received as the result of applying the program \( P_1 \) to the object \( G \). The effect of applying a program of the form \( P \downarrow \) to an object \( G \) is the same as applying the program \( P \) to \( G \) and to the subsequent intermediate results as many times as possible. If \( P \) can’t be applied to \( G \), then the result of applying \( P \downarrow \) to \( G \) is \( G \) itself. In this case it can be thought that \( P \) has been applied to \( G \) zero times. If a program of the form \( \bigcup_i^n P_i \) is applied to an object \( G \), the result is the same as if we had chosen nondeterministically one of the programs \( P_i \) and applied it to the object \( G \).

The effect of applying a program of the form \( \bigcup_i^n P_i \) to an object \( H \), received by applying maximal number of instances of \( P_i \) to \( G \) in parallel. At least one instance of each \( P_i \) should be applied, and there should be no part of the object left, to which any \( P_i \) could be applied. That is, the parallel composition is supposed to work like Lindenmayer systems: everything which is possible to rewrite is rewritten. Besides, some fairness is required too: all programs should be applied at least once. The key question is what criteria programs \( P_i \) should satisfy in order to enable a meaningful definition of the effect of the parallel composition of \( P_i \)'s. Clearly, if for each \( i \) the program \( P_i \) can be applied to an object, then it doesn’t necessarily mean that their parallel composition \( \bigcup_i^n P_i \) can be applied to that object too.

In case of high-level productions such problems have been studied for a long time. A very profound overview of this topic can be found in [24]. Some nice and powerful synchronization techniques for defining parallel application of high-level productions were described there. The essence of these techniques is the following: for each pair of productions define a synchronizing production such that intersections of the images of the left-hand sides of the given pair of productions are changed according to the synchronizing productions. To extend these ideas to high-level replacement programs turned out to be quite difficult and useless for the goals of this work.

The procedure of applying a program \( \bigcup_i^n P_i \) to an object \( G \) could be the following. First, choose a multiset \( S \) of \( P_i \)'s, such that each \( P_i \) is represented in that multiset at least once. Apply the multiset \( S \) to \( G \) in the following manner: each member \( \pi \) of the multiset \( S \) grabs a subobject \( o_\pi \) of the object \( G \). Each element \( \pi \) starts to rewrite its object \( o_\pi \). No object \( o_\pi \) is allowed to contain redundant parts, that is, parts which are never used by any execution of \( \pi \). If for two elements \( \pi \) and \( \pi' \) their objects \( o_\pi \) and \( o_{\pi'} \) do overlap, then \( \pi \) and \( \pi' \) are required not to change anything belonging to the intersection of \( o_\pi \) and \( o_{\pi'} \). If they do, then \( S \) is discarded. If a \( \pi \) is not applicable to its object \( o_\pi \) then \( S \) is discarded. Besides no subobject of the object can be left, that could be used by an instance of some \( P_i \). That is, as many instances of \( P_i \)'s should be executed as possible. If this doesn’t hold, then \( S \) is discarded too. The result of applying
4.4. HIGH-LEVEL REPLACEMENT PROGRAMS (HLRP)

$S$ to $G$ is the object obtained by allowing the elements of $S$ to rewrite their subobjects $o_x$ provided $S$ hasn’t been discarded during the procedure described above. The result of applying $||P_i^n$ to $G$ is defined as an object obtained as a result of applying a certain multiset of $P_i$ to $G$.

The result of applying a program of the form $P = \prod_i^n P_i$ to an object $G$ is the same as if program $||P_i^n$ had been applied, except the case when a program $P_i$ can’t be applied to the object $G$. If for a certain $i = 1 \ldots n$ the program $P_i$ can’t be applied to the object $G$ then the program $||P_i^n$ can still be applied and the result will be the same as if $\prod_{i 
eq j}^n P_j$ had been applied to $G$. If $n = 1$ and $P_1$ can’t be applied to $G$ then $||P_1$ can’t be applied to $G$ as well.

4.4.2 Semantics of high-level replacement programs

To be able to define parallel composition in the manner described above, one needs to characterize execution of a program. Given a program $P$, the execution of $P$ when applied to an object $G$ might be thought as a derivation sequence starting with $G$, where at each step one of productions constituting $P$ is applied. Which productions are applied and in which order is determined by the syntactical structure of $P$ and by the content of the object $G$. Since our programs are supposed to work in non-deterministic manner, application of a program $P$ to an object $G$ might give rise to several derivation sequences. If we forget about parallel composition for a while, then it is easy to see that one can determine all possible executions of a program by decomposing it to simpler programs. Namely, for an elementary program all executions starting from an object $G$ are just one step derivations. For sequential composition of programs $P = P_1; P_2$ an execution of $P$ is a concatenation of executions of $P_1$ and $P_2$. An execution of a program $P \downarrow$ is just a concatenation of executions of the program $P$. An execution of a program of the form $\bigotimes^i P_i$ is simply an execution of the program $P_i$ for some $i = 1 \ldots n$. So it seems to be rather obvious to define the semantics of HLRPs through their possible executions, that is, as a partial mapping from programs and objects to sets of derivation sequences. It turns out, that it is possible to construct the execution of a parallel composition of programs.

A similar approach has been used in [25] where programmed rewriting of relational structures has been defined. Rewriting of relation structures is a special case of high-level replacement. For the description of rewriting systems on relational structures as high-level replacement systems see [13]. In [25] the semantics of programmed rewriting systems was defined by sets of derivation sequences. On contrary, in [14] the semantics of programmed graph transformational systems was defined as a binary relation. Similarly, the semantics of graph transformation units described in [19] was defined as a binary relation too. Both programmed graph transformational systems and graph transformation units allow only sequential control structures. Note that the approach used here makes it possible to define the semantics of programs as binary relations. However, this definition of program semantics as binary relation is based on the notion of program semantics defined via derivation sequences. In the subsequent
text the definition of semantics of HLRPs will be given in a formal way.

For definition of semantics of parallel composition the following function is needed.

**Definition 4.4.2.** Let \( S = \{\tau_1, \tau_2, \ldots, \tau_n\} \) be set of direct derivations with common first element, that is \( G = \text{first}(\tau_i) = \text{first}(\tau_j) \), \( (1 \leq i, j \leq n) \) for some object \( G \). Let \( \tau_i = G \Rightarrow_{p_i, o_i} H_i \). If elements of \( S \) are pairwise parallel independent, then define \( \text{COMP}^P(S) = G \Rightarrow \Sigma_{i=1}^n H_i \) where \( G \Rightarrow \Sigma_{i=1}^n H_i \) is the corresponding parallel composition of derivations from set \( S \). Otherwise \( \text{COMP}^P(S) \) is undefined.

The partial mapping \( \text{COMP}^P(S) \) takes a set of direct derivation as its argument and returns the parallel composition of these derivations if they are parallel independent. Otherwise \( \text{COMP}^P(S) \) is undefined.

Now we are ready to define the semantics of high-level replacement programs formally. The semantics will be given as a mapping \( \sigma \). Mapping \( \sigma \) has two arguments: the first is a program, the second is an object. The value of \( \sigma(P, G) \) is a set of direct derivations. If the value of \( \sigma(P, G) \) is the empty set, it means that the program \( P \) can’t be applied to the object \( G \). Here the execution of a program \( P \) is described by a direct derivation instead of a sequence of derivations. The reason for that is that it is much easier to deal with direct derivations than with sequences of derivations. As it was shown in Subsection 4.3.3 it is possible to construct a direct derivation for each derivation sequence, in such a way that the direct derivation simulates the effect of the derivation sequence. The construction described in Subsection 4.3.3 will be used to transform executions of programs to direct derivations. The formal definition goes as follows:

**Definition 4.4.3 (Semantics of high-level replacement programs).** Let \( CAT \) be a category and \( M \) a class of arrows. The semantics of a well-formed high-level replacement program \( P \) denoted by \( \sigma(P) \) is a mapping from objects of \( CAT \) to sets of direct derivations over \( CAT \). The mapping defined recursively as follows:

- (i) If \( P = \{p_1, \ldots, p_n\} \) is a finite set of rules from \( P \) then
  \[
  \sigma(P)(G) = \{ G \Rightarrow_{p, o} H | p \in P \text{ and } p \text{ is applicable to } G \text{ through } o \}
  \]

- (ii) If \( P = P_1; P_2 \) then
  \[
  \sigma(P)(G) = \{ \text{PROD}(\tau_1 \circ \tau_2) | \tau_i \in \sigma(P_i)(G), i = 1, 2 \}
  \]

- (iii) If \( P \) is a production rule \( \tau \) then
  \[
  \sigma(P)(G) = \{ \text{PROD}(\tau_1 \circ \ldots \circ \tau_n) | (\forall 2 \leq i \leq n : \tau_i \in \sigma(P')(\text{last}(\tau_{i-1})) \wedge \tau_n \in \sigma(P')(G) \land \sigma(P')(\text{last}(\tau_n)) = \emptyset \}
  \]
  \[
  \cup \{ G \Rightarrow_{p, o} G | p \notin G \text{ if } \sigma(P')(G) = \emptyset \}
  \]
4.4. **HIGH-LEVEL REPLACEMENT PROGRAMS (HLRP)**

- (iv) If $P = ||^n P_i$ then
  \[
  \sigma(P)(G) = \{ COMP'(S) \mid S \subseteq \bigcup^n_i \sigma(P_i)(G) \land \\
  (\forall 1 \leq i \leq n : S \cap \sigma(P_i)(G) \neq \emptyset) \\
  \land (\exists S' \in \bigcup^n_i \sigma(P_i)(G) : \\
  S \subset S' \land COMP'(S') \text{ is defined} ) \\
  \land COMP'(S) \text{ is defined} \}
  \]

- (v) If $P = \prod^n_i P_i$ then
  \[
  \sigma(P)(G) = \left\{ \begin{array}{ll}
  \sigma(\prod^n_{j \neq i} P_j)(G) & \text{if } \exists i : \sigma(P_i)(G) = \emptyset \text{ and } n > 1 \\
  \sigma(\prod^n_i P_i)(G) & \text{otherwise}
  \end{array} \right.
  \]

- (vi) If $P = \bigoplus^n_i P_i$ then
  \[
  \sigma(P)(G) = \bigcup^n_i \sigma(P_i)(G)
  \]

The definition of semantics of elementary programs, iteration, sequential composition and non-deterministic composition of programs is rather obvious. The definition of semantics of parallel composition is much more involved. As it was already mentioned, direct derivations correspond to executions of programs. The set $S$ in the definition of the semantics of parallel program composition corresponds to the multiset of instances of $P_i$s. Indeed, each application of an instance of $P_i$ to $G$ gives rise to several executions, that is, to several direct derivations. For each direct derivation $\tau_i$ the left-hand side of the production $p_i$ used in $\tau_i$ corresponds to the subobject grabbed by the instance of some $P_j$ corresponding to $\tau_i$. Naturally, the gluing object of the production $p_i$ corresponds to that part of the subobject, which is read but isn’t changed by the program. So, the parallel independence of direct derivations constituting $S$ corresponds to the requirement that the overlapping parts of the subobjects must not be changed. The parallel composition of direct derivations belonging to $S$ gives exactly the effect of applying the set of instances corresponding to the direct derivations constituting $S$ to the object $G$. So, the semantics defined above indeed meets our requirements.

The semantics of HLRPs given above is well defined for HLRP categories. Formally, the following theorem holds

**Theorem 4.4.1.** Assume that category $CAT$ with the class of arrows $M$ satisfies HLRP conditions. Then the semantics of high-level replacement programs given in Definition 4.4.3 is well defined. That is, for each well-formed high-level replacement program $P$ over $CAT$ and for each object $G$ of $CAT$ the value of $\sigma(P)(G)$ is well defined. Besides, the following holds:

\[
\tau \in \sigma(P)(G) \implies \text{firs}(\tau) = G \quad (4.8)
\]
Proof. The proof goes by structural induction on $P$.

- If $P$ is an elementary program, then $\sigma(P)(G)$ is clearly well defined for each object $G$. Besides $\text{first}(\tau) = G$ for each $\tau \in \sigma(P)(G)$.

- Assume that for program $P_1$ and $P_2$ the statement of the theorem holds. For each $\tau_1 \in \sigma(P_1)(G)$ and $\tau_2 \in \sigma(P_2)(\text{last}(\tau_1))$ we have by (4.8) that $\tau_1$ and $\tau_2$ composable. From the assumption that CAT with class of arrows $M$ is a HLRP category by using Theorem 4.3.2 we get that $\text{PROD}(\tau_1 \circ \tau_2)$ is well defined. Besides, it is clear from Definition 4.3.5 that $\text{first}(\text{PROD}(\tau_1 \circ \tau_2)) = \text{first}(\tau_1 \circ \tau_2) = G$. Thus we get that the statement of the theorem holds for the program $P_1; P_2$

- Assume that for the program $P$ the statement of the theorem holds. Then similarly to the proof of the previous case it can be proven that the statement of theorem holds for $P \downarrow$.

- Assume that for each $i = 1 \ldots n$ the statement of the theorem is true for $P_i$. Then it will be true for program $P = \parallel^n P_i$ as well. Indeed, for each object $G$ the set $\sigma(P_i)(G)$ is well defined, which means that $S \subseteq \bigcup_i \sigma(P_i)(G)$ is well defined. This implies that $\sigma(P)(G)$ is well defined. On the other hand, for each $\tau = \text{COMP}'(S)$ for some $S \subseteq \bigcup_i \sigma(P_i)(G)$ we have that $\text{first}(\tau) = \text{first}((\text{COMP}'(S))) = \text{first}(\omega) = G$ for each $\omega \in S$. That is, the statement of the theorem holds for $P$ as well.

- Assume that for each $P_i$ the statement of the theorem holds. Then it holds for $P = \parallel^n P_i$ as well. The proof goes by induction on $n$. For $n = 1$, $\sigma(\parallel P_i)(G) = \sigma(\parallel P_i)(G)$ for each object $G$, therefore, by the previous case the statement of the theorem holds for $P$. For $n > 1$ and for each $G$ two cases are possible. Either $\sigma(P_j)(G) \neq \emptyset$ for all $j = 1 \ldots n$, or $\sigma(P_j)(G) = \emptyset$ for some $j = 1 \ldots n$.

In the former case $\sigma(P)(G) = \sigma(\parallel^n P_i)(G)$, which implies that $\sigma(P)(G)$ is well defined and (4.8) holds for it. In the latter case we have that $\sigma(P)(G) = \sigma(\bigcup P_i)(G)$ which implies by induction hypothesis that $\sigma(P)(G)$ is well defined and (4.8) holds for it. That is, the statement of the theorem holds for $P$.

- Assume that for each $i = 1 \ldots n$ the statement of the theorem holds for $P_i$. Then it is easy to see that it holds for $P = \bigcup^n P_i$ as well.

\[ \square \]

Using the definition of program semantics given above it is possible to define program semantics as a binary relation $\Longrightarrow$. Namely:

\[ G \Longrightarrow_P H \iff 3\tau = G \Longrightarrow H : \tau \in \sigma(P)(G) \]

In some cases treating programs as binary relations might be more useful. Particularly, it is more convenient to treat programs as binary relations when defining abstract networks of rewriting systems.
Chapter 5

Abstract networks of rewriting systems

5.1 Introduction

The aim of this chapter is to introduce a generalization of NLP systems described in Chapter 3. Recall that NLP systems are systems consisting of several string rewriting systems (or language processors) which cooperate with each other by exchanging strings. A very obvious extension of NLP systems would be allowing rewriting of other objects, not just strings. For example, by allowing rewriting of graphs it would be possible to study more recent models of DNA computers within the framework of NLP systems. In this chapter a formal description of this generalized version of NLP systems will be given. To be able to create such a formalism the following two questions should be answered:

- How to describe the communication of rewriting systems in a uniform way?
- How to describe different ways of rewriting by a uniform formalism?

A possible answer to the first question is to use graph rewriting to encode communication. Graphs and graph rewriting systems have been used for modeling multi-agent systems for long time. See [18, 16, 17] for more information on this approach. In the approach introduced in this work the communication of generalized NLP systems will be encoded by programmed rewriting of the graph that describes the communication of the system. The solution of the problem raised by the second question is to use categories to describe the class of objects which are to be rewritten and encode rewriting by high-level replacement productions.

To sum up, communication will be encoded by series of graph transformations, rewriting of data objects by series of high-level production applications. The state of the system will be described as hierarchical structure consisting of a
\textit{communication graph} – graph describing the state of interaction of agents and a multiset of data objects, such that each member of the multiset is assigned to one of the agents of the system. A state transition of the system will be generated by applying a high-level replacement program to the object encoding the current state of the system. State transitions of two types will be distinguished

- During a \textit{communication step} both the \textit{communication graph} and the data objects are changed. This step corresponds to the event when agents communicate with each other.
- During a \textit{rewriting step} only data objects are changed. This step corresponds to rewriting steps of NLP systems. This step is performed by parallel application of programs that encode rewriting performed by agents.

In this chapter the formalization of the concepts formulated above will be given. Section 5.2 contains the abstract definition of generalized NLP systems. We will refer to this construction as \textit{abstract network of rewriting systems} (ANRS for short). The expressive power of ANRS systems will be demonstrated by formalizing NLP systems introduced in Chapter 3 within the framework of ANSR systems. This will be done in Section 5.3.

\section*{5.2 Abstract networks of rewriting systems}

In this section the notion of abstract networks of rewriting systems (ANRS for short) will be introduced. Subsection 5.2.1 contains the construction of the class of categories used to describe states of an ANRS system. Subsection 5.2.2 contains description of some extensions of HLRP syntax and semantics needed for defining ANRS systems. Finally, the last part of the section, Subsection 5.2.3 contains the formal definition of abstract networks of rewriting systems.

\subsection*{5.2.1 Category of ANRS states}

Let $D$ be a category and let $M_D$ be a class of arrows. Let $TG$ be a directed graph, called the \textit{type graph}. Let $Ag$ be a subgraph of $TG$, called the \textit{agent graph}. We want to define a category, objects of which could be used for describing states of ANRS systems. We suggest that each object of the category should consist of two parts:

- A graph typed over $TG$. The graph is intended to encode the state of the system from the point of view of communication. Since $Ag \subseteq TG$, some nodes of the graph are labeled by agent types. These nodes correspond to the agents of the system. The first part will be called the \textit{communication graph} or \textit{communication configuration}.

- A mapping from the set of nodes of the communication graph labeled by agent types to the class of objects of $D$. For each agent of the system this mapping tells the data object the agent is working on. This part of
the object encodes the state of the system from the point of view of data objects of agents. We will call this mapping the data configuration of the system.

Now the formal definition will be given.

**Definition 5.2.1.** The category of ANRS states with type graph $TG$, agent graph $Ag \subseteq TG$ and data category $D$ denoted by $N_{D,TG,Ag}$ is a category such that

- Objects are of the form $(G, m : G \rightarrow TG, f : V_{m^{-1}(Ag)} \rightarrow Obj(D))$ where $G$ is a directed graph called the communication graph component of the object, $m$ is a graph morphism, $f$ is a mapping from the set of vertices of the inverse image of $Ag$ generated by $m$ to objects of the category $D$ (denoted by $Obj(D)$). The component $f$ is called the data configuration component of the object.

- An arrow between objects $(G, m : G \rightarrow TG, f : V_{m^{-1}(Ag)} \rightarrow Obj(D))$ and $(\hat{G}, \hat{m} : \hat{G} \rightarrow TG, \hat{f} : V_{\hat{m}^{-1}(Ag)} \rightarrow Obj(D))$ is a 2-tuple

$$\langle h, \{h_v | v \in V_{\hat{m}^{-1}(Ag)} \} \rangle$$

such that $h : G \rightarrow \hat{G}$ is an injective graph morphism such that $\hat{m} \circ h = m$ and for all $v \in V_{m^{-1}(Ag)}$, $h_v : f(v) \rightarrow \hat{f}(h(v))$ is an arrow of $D$.

- The identity arrow is

$$\langle id, \{id_f | v \in V_{m^{-1}(Ag)} \} \rangle$$

- The composition of two arrows $(h, \{h_v | v \in V_{m^{-1}(Ag)} \})$ and $(l, \{l_v | v \in V_{\hat{m}^{-1}(Ag)} \})$ is the arrow $(l \circ h, \{l_v \circ h_v | v \in V_{m^{-1}(Ag)} \})$

The construction in Definition 5.2.1 indeed defines a category:

**Theorem 5.2.1.** The construction $N_{D,TG,Ag}$ is indeed a category.

The proof of this theorem is given in Appendix A.

To make the usage of $N_{D,TG,Ag}$ categories easier, the following notation will be used. If $A$ is an object of the category $N_{D,TG,Ag}$, then we will assume that $A$ is of the form $(G_A, m_A : G_A \rightarrow TG, f : V_{m_A^{-1}(Ag)} \rightarrow Obj(D))$. Sometimes objects of the category $N_{D,TG,Ag}$ will be denoted by $(G, m, f)$. $(G, m, f)$ is simply the object $(G, m : G \rightarrow TG, f : V_{m^{-1}(Ag)} \rightarrow Obj(D))$. An arrow $h : A \rightarrow B$ will be assumed to have the following shape $h = (h_G, \{h_v | v \in V_{m_A^{-1}(Ag)} \})$ where $h_G : G_A \rightarrow G_B$ such that $m_B \circ h_G = m_A$ and $h_v : f_A(v) \rightarrow f_B(h_G(v))$ are arrows in $D$.

**Remark 5.2.1.** Consider the following category. Let pairs of the form $(G, m)$ be the objects of the category where $G$ is a directed graph and $m : G \rightarrow TG$ is a graph morphism. An arrow $f : (G, m_G) \rightarrow (H, m_H)$ is such a graph morphism $f : G \rightarrow H$ that $m_H \circ f = m_G$. It is easy to verify that the construction
described above is indeed a category. This category will be called the category of typed graphs over the type graph $TG$. It is easy to verify that the category of typed graphs satisfies HLRP conditions and has all finite limits and colimits.

In fact, for each object $(G, m, f)$ of the category $N_{D, TG, A_g}$, $(G, m)$ is an object of the category of typed graphs over $TG$. Besides, for each arrow $h: A \to B$ of the category $N_{D, TG, A_g}$ the arrow $h_G: G_A \to G_B$ is an arrow of the category of typed graphs over the type graph $TG$. That is, we can write $h_G: (G_A, m_A) \to (G_B, m_B)$. This interpretation of the communication graph part of objects of $N_{D, TG, A_g}$ will be used in the subsequent text.

**Remark 5.2.2.** It is easy to see that two objects $A$ and $B$ of the category $N_{D, TG, A_g}$ are isomorphic if and only if $(G_A, m_A) \cong (G_B, m_B)$ and $f_A(v) \cong f_B(\phi_G(v))$ for each $v \in V_{m^{-1}(A_g)}$ where $\phi_G: G_A \to G_B$ is the isomorphism between $(G_A, m_A)$ and $(G_B, m_B)$.

Category $N_{D, TG, A_g}$ has some very pleasant properties. Namely, if the category $D$ with the class of morphisms $M_D$ satisfies HLRP conditions then so does the category $N_{D, TG, A_g}$. This property is essential for our aim, since we will formalize computation steps of ANRS systems as effect of application of certain HLRPs to the states of the system, that is, to objects of a category $N_{D, TG, A_g}$. More precisely the following theorem holds.

**Theorem 5.2.2.** Assume that a category $D$ with class of morphisms $M_D$ satisfies HLRP conditions. Consider the category $N_{D, TG, A_g}$ where $TG$ is an arbitrary type graph and $A_g \subseteq TG$ is an arbitrary agent graph. Define class $M$ of arrows of $N_{D, TG, A_g}$ as the class of arrows $(h, \{h_v | v \in V_{m^{-1}(A_g)} \})$ such that $h_v$ are in $M_D$ for all $v \in V_{m^{-1}(A_g)}$. Then the category $N_{D, TG, A_g}$ together with class of arrows $M$ satisfies HLRP conditions

The proof of this theorem is given in Appendix A.

The definition of $N_{D, TG, A_g}$ categories was inspired by hierarchical graphs defined in [21]. It is worth to note that construction in Definition 5.2.1 allows creating categories which resemble categories of hierarchical hypergraphs of a certain depths.

### 5.2.2 Extension of syntax and semantics of high-level replacement programs

In order to be able to define ANRS systems, an extension of the syntax and semantics of high-level replacement programs is needed. Namely, we would need to define how to transform high-level replacement programs defined over a certain category $D$ to a HLRP defined over the category $N_{D, TG, A_g}$ while preserving the semantics. This transformation is essential because we will describe rewriting performed by agents on their data objects as application of HLRPs defined over the category of data objects.

The operation on programs described below will be called the hierarchical extension of programs. Consider a program $P$ over $D$. Let be $a \in V_{A_g}$. Then define $H_aP$ to be a program over $N_{D, TG, A_g}$, such that if applied to an object
\[ (G, m : G \rightarrow TG, f : V_{m^{-1}(A)} \rightarrow \text{Obj}(D)) \text{ in } N_{D, TG, A_d} \] it would apply the program \( P \) to an object \( f(v) \) such that \( m(v) = a \). To define the semantics of \( H_aP \) some preliminary definitions will be needed.

**Definition 5.2.2.** Let \( p = (L \xleftarrow{\ell} K \xrightarrow{r} R) \) be a high-level replacement production over category \( D \) with class of morphisms \( M_p \). That is, \( \ell, r \) are in \( M_D \). Let \( a \in V_{A_d} \). Then define \( H_aP \) as the following production over \( N_{D, TG, A_d} \):

\[
H_aP_i = H_aL \xleftarrow{\ell} H_aK \xrightarrow{r} H_aR
\]

such that

\[
H_aL = (G_a, m_a, f_L) \text{ where } f_L(v_a) = L \\
H_aK = (G_a, m_a, f_K) \text{ where } f_K(v_a) = K \\
H_aR = (G_a, m_a, f_R) \text{ where } f_R(v_a) = R
\]

where \( G_a = (v_a, \emptyset, \emptyset, \emptyset), m_a(v_a) = a \) and \( \ell = (id_{G_a}, \{\ell\}) \) and \( r = (id_{G_a}, \{r\}) \).

It is easy to see that the following statement holds.

**Claim 5.2.1.** If class \( M \) of arrows of category \( N_{D, TG, A_d} \) is defined as in Theorem 5.2.2 then for each \( a \in V_{A_d} \) and for each high-level replacement production \( p \) over \( D \) the construction \( H_aP \) is a high-level replacement production over \( N_{D, TG, A_d} \).

The following theorem describes the effect of applying a production \( H_aP \) to an object of the category \( N_{D, TG, A_d} \). We will assume, that the category \( N_{D, TG, A_d} \) with class of arrows \( M \) has been constructed as described in Theorem 5.2.2.

**Theorem 5.2.3.** Let \( p = (L \xleftarrow{\ell} K \xrightarrow{r} R) \) be a production over \( D \). Let \( (G, m, f) \) be an object of \( N_{D, TG, A_d} \). Then \( H_aP \) is applicable to \( (G, m, f) \) via \( \alpha = (\alpha_G, \{\alpha_a\}) : H_aL \rightarrow (G, m, f) \) if and only if \( p \) is applicable to \( f(\alpha_G(v_a)) \) through \( \alpha_a \). Besides, if \( f(\alpha_G(v_a)) \Rightarrow_{\alpha, \alpha_a} z \) holds, then \( (G, m, f) \Rightarrow_{H_aP, \alpha} (G, m, f_H) \) holds, where \( f_H(v) = \begin{cases} \text{if } v = \alpha_G(v_a) \\ f(v) \text{ otherwise} \end{cases} \)

**Proof.** Consider the following diagram

\[
\begin{array}{c}
H_aL \\
H_aK \\
H_aR
\end{array}
\]

\[
\begin{array}{c}
\ell \\
r \\
t
\end{array}
\]

\[
\begin{array}{c}
\Rightarrow_{\alpha, \alpha_a} z \\
(1) \\
(2)
\end{array}
\]

\[
\begin{array}{c}
(G, m, f) \\
(D, m_D, f_D)
\end{array}
\]

\[
\begin{array}{c}
\Rightarrow_{H_aP, \alpha} (H, m_H, f_H)
\end{array}
\]

where \( h = (\alpha_G, \{h_{v_a}\}) \) and \( t = (\alpha_G, \{t_{v_a}\}) \). Assume that diagrams (1) and (2) are pushouts. From the colimit construction for \( N_{D, TG, A_d} \) (see Appendix A) it follows that diagrams (1) and (2) are pushouts if and only if each of the following conditions holds:
• \((G, m) = (D, m_D) = (H, m_H)\)

• \(g = \{id_G, \{d_{o_G(v_a)}\} \cup \{d_{f(v)}| v \in V_{m^{-1}(A_g)} \land v \neq o_G(v_a)\}\} \) and \(d = \{id_G, \{d_{o_G(v_a)}\} \cup \{d_{f(v)}| v \in V_{m^{-1}(A_g)} \land v \neq o_G(v_a)\}\}\)

• \(f_H(v) = f(v) = f_D(v)\) for all \(v \in V_{m^{-1}(A_g)} \setminus \{o_G(v_a)\}\)

• Diagrams (3) and (4) below are pushouts or, in other words, the direct derivation \(f(o_G(v_a)) \implies p, z\) is of the form

\[
\begin{array}{c}
L \quad l \quad K \quad r \quad R
\end{array}
\]

\[
\begin{array}{c}
\uparrow \\
o_G(v_a)
\end{array}
\quad (3)
\quad \uparrow h_{v_a}
\quad (4)
\quad \uparrow t_{v_a}
\]

\[
\begin{array}{c}
f(o_G(v_a))
\quad \frac{g_{o_G(v_a)}}{f_D(o_G(v_a))}
\quad \frac{d_{o_G(v_a)}}{f_H(o_G(v_a))}
\end{array}
\]

That is, the statement of the theorem holds.

The theorem says that applying \(H_{aP}\) to an object \(O = (G, m, f)\) has the same effect as applying \(p\) to the data belonging to an agent of the object \(O\) with agent type \(a\) and replacing the data object of the agent with the result of application of \(p\) to the original data object of the agent.

To be able to define the semantics of \(H_aP\) in terms of sets of direct derivations the following definition will be needed.

**Definition 5.2.3.** Let \(\tau = g \implies_{p, o} z\) be a direct derivation over \(D\). Let \((G, m, f)\) be an object of \(N_{D,TG,A_g}\). Let \(a \in V_{A_g}\) be an agent type. Assume that there exists \(v \in V_{m^{-1}(a)}\) such that \(f(v) = g\). Define

\[H_{(G,m,f), a}(\tau) := \{(G, m, f) \implies_{H_{aP}, o} (G, m, f')|H_{aP} = (o', \{o\}) \land f(o'(v_a)) = g\}\]

If for all \(v \in V_{m^{-1}(a)} f(v) \neq g\) holds, then define \(H_{(G,m,f), a}(\tau)\) to be the empty set. We will call \(H_{(G,m,f), a}(\tau)\) the extension of \(\tau\) to category \(N_{D,TG,A_g}\).

Extensions of direct derivations to \(N_{D,TG,A_g}\) have some nice and important properties. They are collected in the theorem below.

**Theorem 5.2.4.** Let \(\tau = g \implies_{p} z\) be a direct derivation over \(D\). Let \((G, m, f)\) be an object of \(N_{D,TG,A_g}\) and let \(a\) be an agent type \((a \in V_{A_g})\). Then

• (a) \(H_{(G,m,f), a}(\tau)\) is well defined

• (b) \(\forall \omega \in H_{(G,m,f), a}(\tau) : first(\omega) = (G, m, f)\)

• (c) If \(\omega = (G, m, f) \implies_{H_{aP}, o} (G, m, f')\) is a direct derivation over \(N_{D,TG,A_g}\) then there exists a direct derivation \(\omega' = f(o_G(v_a)) \implies_p f'(o_G(v_a))\) over \(D\) such that \(\omega \in H_{(G,m,f), a}(\omega')\).
5.2. ABSTRACT NETWORKS OF REWRITING SYSTEMS

Proof. All three statements of the theorem are corollaries of Theorem 5.2.3. □

Now formal definition of syntax and semantics of the extended extended version of high-level replacement programs will be given. The extended version of high-level replacement programs will be abbreviated by ext-HLRP. The syntax and semantics of ext-HLRPs is essentially the same as that of HLRPs except that hierarchical extension can be used for constructing valid ext-HLRP programs too.

Definition 5.2.4 (Syntax of ext-HLRP). Consider a category $CAT$ and class of morphisms $M$. Let $\mathcal{P}$ be a set of high-level replacement productions over $CAT$. $P$ is a well-formed extended high-level replacement program over category $CAT$ if and only if it arises by applying one of the following rules:

- $P = \mathcal{P}' \subseteq \mathcal{P}$, provided that $\mathcal{P}'$ is finite, is a well-formed ext-HLRP. Such programs are called elementary programs.
- If $P_1$ and $P_2$ are well-formed ext-HLRPs, so is their sequential composition $P_1 ; P_2$.
- If $P$ is a well-formed ext-HLRP, so is its iteration $P\downarrow$.
- If $P_1, \ldots, P_n$ are well-formed ext-HLRPs, so is their parallel composition $|||P_i$.
- If $P$ is a well-formed ext-HLRP over category $D$ and $CAT = N_{D,TG,A_g}$ for some graphs $A_g, TG$ and $a \in V_{A_g}$ then $H_n P$ is a well-formed ext-HLRP over category $CAT$.
- If $P_1, \ldots, P_n$ are well-formed ext-HLRPs then so is their weak parallel composition $\lceil\lceil P_i$.
- If $P_1, \ldots, P_n$ are well-formed ext-HLRPs then so is their non-deterministic composition $\bigcirc_i^n P_i$.

The semantics of ext-HLRPs is an extension of the semantics of HLRPs. Similarly to the semantics of HLRPs the semantics of ext-HLRPs is defined as a partial mapping from programs and objects to sets of direct derivations. For elementary programs and programs of the form $P_1 ; P_2$ and $|||P_i$, $\lceil\lceil P_i$ and $\bigcirc_i^n P_i$ the semantics of ext-HLRPs coincides with the semantics of HLRPs.

Definition 5.2.5 (Semantics of ext-HLRPs). Let $P$ be an ext-HLRP program over category $CAT$ with class of morphisms $M$. Then the semantical mapping $\sigma(P)$ is defined recursively on the structure of $P$:

- (i) If $P = \{ p_1, \ldots, p_n \}$ is a finite set of rules from $\mathcal{P}$ then

  $$\sigma(P)(G) = \{ G \Rightarrow H | p \in P \text{ and } p \text{ is applicable to } G \text{ through } o \}$$
(ii) If $P = P_1; P_2$ then
\[
\sigma(P)(G) = \{\text{PROD}(\tau_1 \circ \tau_2) | \tau_i \in \sigma(P_i)(G), i = 1, 2\}
\]

(iii) If $P = P' \downarrow$ then
\[
\sigma(P)(G) = \{\text{PROD}(\tau_1 \circ \tau_2 \cdots \circ \tau_n) | (\forall 2 \leq i \leq n : \tau_i \in \\
\sigma(P') (\text{last}(\tau_{i-1})) \land \tau_1 \in \sigma(P')(G) \land \\
\sigma(P')(\text{last}(\tau_n)) = \emptyset) \\
\cup \{G \Rightarrow_{p_G} G\} \\
p_G = (G \uparrow G G \downarrow G \text{ if } \sigma(P')(G) = \emptyset\}
\]

(iv) If $P = \llbracket^n P_i$ then
\[
\sigma(P)(G) = \{\text{COMP}'(S) | S \subseteq \bigcup^n \sigma(P_i)(G) \land \\
(\forall 1 \leq i \leq n : S \cap \sigma(P_i)(G) \neq \emptyset) \\
\land (\neg \exists S' \in \bigcup^n \sigma(P_i)(G) : \\
S \subseteq S' \land \text{COMP}'(S') \text{ is defined} ) \\
\land \text{COMP}'(S) \text{ is defined } \}
\]

(v) If $P = H_a P'$ then
\[
\sigma(P)(G) = \{\omega | \exists g \in \text{Obj}(D) : \exists \tau \in \sigma(P')(g) : \omega \in H_{G,a}(\tau)\}
\]

(vi) If $P = \llbracket^n P_i$ then
\[
\sigma(P)(G) = \begin{cases} 
\sigma(\llbracket^n P_i \downarrow P_j)(G) & \text{if } \exists i : \sigma(P_i)(G) = \emptyset \text{ and } n > 1 \\
\sigma(\llbracket^n P_i)(G) & \text{otherwise}
\end{cases}
\]

(vii) If $P = \bigodot^n P_i$ then
\[
\sigma(P)(G) = \bigcup^n \sigma(P_i)(G)
\]

As in the case of HLRPs, if $\sigma(P)(G) = \emptyset$ for some ext-HLRP $P$ and some object $G$ then it means that program $P$ is not applicable to the object $G$. From the definition above it immediately follows that the semantics of ext-HLRPs is indeed an extension of the semantics of HLRPs an HLRP program viewed as an ext-HLRP program has the same semantics as if it was considered to be a HLRP program. It is easy to see that the following lemma is true.

**Lemma 5.2.1.** Assume that $\text{CAT}$ together with class of arrows $M$ satisfies HLRP conditions. Assume that $P$ is an arbitrary ext-HLRP over $\text{CAT}$ such
5.2. ABSTRACT NETWORKS OF REWRITING SYSTEMS

that $P$ doesn't contain hierarchical extension. That is, $P$ is either an elementary program or has been constructed from elementary programs by using iteration, sequential composition, parallel composition, weak parallel composition and non-deterministic composition of programs. Then for all objects $G$ of CAT $\sigma(P)(G)$ is well defined, besides

$$\forall \tau \in \sigma(P)(G) : \text{first}(\tau) = G$$

**Proof.** The proof is analogous to the proof of Theorem 4.4.1. □

The following theorem and its corollary ensures that extended high-level replacement programs are suitable for our purposes.

**Theorem 5.2.5.** Assume that $D$ with class of arrows $M_D$ satisfies HLRP conditions and for each ext-HLRP $P'$ over $D$ and for each object $z$ of $D$ the set $\sigma(P')(z)$ is well defined. Then for each ext-HLRP $P$ over $N_{D,TG,A_9}$ with class of arrow $M$ defined as in Theorem 5.2.2 and for each object $G$ of $N_{D,TG,A_9}$ the set $\sigma(P)(G)$ is well defined and

$$\forall \tau \in \sigma(P)(G) : \text{first}(\tau) = G$$

**Proof.** According to Theorem 5.2.2 $N_{D,TG,A_9}$ together with class of arrows $M$ satisfies HLRP conditions. Besides, because of the assumption that $\sigma(P')(z)$ is well defined for all ext-HLRPs $P$ over $D$ and objects $z$ of $D$ by using Theorem 5.2.4 we get that $\sigma(H,P')(G)$ is well defined. Besides, for each $\tau \in \sigma(H,P')(G)$ we have that $\text{first}(\tau) = G$. The proof of the rest of the theorem goes analogously to the proof of Theorem 4.4.1. □

The following statement is a corollary of Theorem 5.2.5

**Corollary 5.2.1.** Assume that $D$ together with class of arrows $M_D$ satisfies HLRP conditions and $D$ is not of the form $N_{D',TG',A_9'}$. Consider ext-HLRPs over $N_{D,TG,A_9}$ with class of arrows $M$ defined as in Theorem 5.2.1. Then for each ext-HLRP $P$ over $N_{D,TG,A_9}$ and for each object $G$ of $N_{D,TG,A_9}$ the set $\sigma(P)(G)$ is well defined and

$$\forall \tau \in \sigma(P)(G) : \text{first}(\tau) = G$$

**Proof.** Since $D$ is not of the form $N_{D',TG',A_9'}$ we have that all ext-HLRPs over $D$ do not contain hierarchical extension. By Lemma 5.2.1 it means that for all ext-HLRPs $P'$ over $D$ $\sigma(P')(z)$ is well defined for each object $z$ of $D$. By Theorem 5.2.5 it means that for each ext-HLRP $P$ over $N_{D,TG,A_9}$ and for each object $G$ of $N_{D,TG,A_9}$ $\sigma(P)(G)$ is well defined and $\forall \tau \in \sigma(P)(G) : \text{first}(\tau) = G$. □

Theorem 5.2.5 and Corollary 5.2.1 have the following important consequence: if we start from a HLRP category $D$ and we keep on constructing ANRS state categories according to Theorem 5.2.2 then we can always use extended high-level replacement programs, since their semantics will always be well defined.
Notice that the semantics of hierarchical composition indeed meets the requirement formulated in the beginning of this section. $H_a P$ indeed acts exactly as if $P$ was applied to the data object of an agent of type $a$. More precisely, the following statement is true.

**Theorem 5.2.6.** Let $P$ be a program over $D$. Let $a \in V_{Ag}$ be an agent type. Then $(G, m, f) \Rightarrow_{H_a P} (H, m_H, f_H)$ if and only if $(G, m) = (H, m_H)$ and there exists a vertex $w \in m^{-1}(a)$ such that $f(w) \Rightarrow f_H(w)$ and $f(v) = f_H(v)$ for all $w \neq v \in V_{m^{-1}(Ag)}$.

**Proof.** $(G, m, f) \Rightarrow_{H_a P} (H, m_H, f_H)$ holds if and only if there exists a direct derivation $\omega = (G, m, f) \Rightarrow \sigma \ (H, m_H, f_H)$ such that $\omega \in \sigma(H_a P)((G, m, f))$. This means that $\omega \in H_{(G, m, f), a}(\tau)$ for some $\tau \in \sigma(P)(f(\alpha_G(v_a)))$. From definition of $H_{(G, m, f), a}(\tau)$ it follows that $p = H_a \sigma \tau$ and $\tau = f(\alpha_G(v_a)) \Rightarrow \sigma \tau \ f_H(\alpha_G(v_a))$. That is, by choosing $w = \alpha_G(v_a)$ we get that $f(w) \Rightarrow f_H(w)$ and $f(v) = f_H(v)$ for all $w \neq v \in V_{m^{-1}(Ag)}$. \hfill \blacksquare

### 5.2.3 Abstract networks of rewriting systems

Now we are ready to define the notion of abstract network of rewriting systems. We will assume that agents are of certain type. Each type corresponds to a vertex of the agent graph, or, in other words, there is a bijection between the set of types and the set of vertices of the agent graph. Edges of the agent graph are intended to describe certain relationships between different types. The communication is encoded by directed graphs typed over the type graph $TG$. The agent graph $Ag$ is assumed to be a subgraph of the graph $TG$. In this work agent graphs will be always discrete graphs. In fact, because of the requirement that the agent graph is a subgraph of the type graph, it is sufficient to take a discrete graph as the agent graph, since the relationships between agents can be encoded by the type graph. On the other hand, the possibility of encoding such relationships by edges of the agent graph allows more general treatment without making the constructions more complex.

Agents are assumed to do rewriting over the objects of a certain category $D$. We will assume that not necessarily all agents are important from the point of view of the systems but just agents of certain types. That is, agents can be of a passive type. Agents of passive types are used to make the description of the communication process easier. For example they might be used as intermediate data storage during the communication process. Agents of some other types are assumed to be always important for the description of the whole system. Their make up the behavior of the whole system and are of particular interest. Such agent types will be called active agent types and an agent of such a type will be called active agent. Only data objects of active agents will be observed during the work of the system. That is, active agents correspond to the agents of NLP systems: active agents do perform rewriting on data objects.

States of the system will be objects of $N_{D,TG,Ag}$. We will assume that $D$ together with class of morphisms $M_D$ satisfies HLRP properties and the
semantics of extended high-level replacement programs over $D$ is well defined. We will construct class $M$ of arrows of $N_{D,TG,A_g}$ as described in Theorem 5.2.2. Then by the results of Section 5.2.2 we can use extended high-level replacement programs over $N_{D,TG,A_g}$. It will be assumed that each active agent performs rewriting in a way specific to its type: we will associate an ext-HLRP over $D$ with each active agent type. Agents of a certain active agent type perform rewriting by applying the ext-HLRP associated with their types to their data objects. The effect of communication is defined as the effect of applying a certain ext-HLRP over $N_{D,TG,A_g}$ to the current state of the system.

Then for abstract network of rewriting systems the following definition might be given.

**Definition 5.2.6.** An abstract network of rewriting systems (ANRS for short) is the following tuple

$$
\Gamma = (TG, A_g, D, S_0, A_{g_{act}}, P_{\text{comm}}, \{P_a | a \in V_{A_{g_{act}}})
$$

where

- $TG$ is the type graph of the communication graphs.
- $A_g \subseteq TG$ is the agent graph of the system, $A_g$ is a subgraph of $TG$.
- $A_{g_{act}} \subseteq A_g$ is the graph of active agent types, $A_{g_{act}}$ is a subgraph of $A_g$.
- $D$ is the category of data objects of the system.
- $S_0$ is the initial state (or initial configuration) of the system, $S_0$ is an object of $N_{D,TG,A_g}$.
- $P_a$ are the ext-HLRPs over $D$ which define the way how agents of type $a$ perform rewriting on their data objects.
- $P_{\text{comm}}$ is the ext-HLRP over $N_{D,TG,A_g}$ which describes the communication of the agents of the system.

Objects of $N_{D,TG,A_g}$ are used for describing configurations of the system $\Gamma$. $P_{\text{comm}}$ is the program which does the communication step. Programs $P_a$ ($a \in V_{A_{g_{comm}}}$) are used during the rewriting step. That is, using standard notations established in the area of NLPs:

$$
\xrightarrow{\text{comm}} := \xrightarrow{P_{\text{comm}}}
$$

and

$$
\xrightarrow{\text{rew}} := \big|_{a \in V_{A_{g_{act}}} \text{ } H_a P_a}
$$

The main difference between a rewriting step and a communication step is the following. Programs performing a rewriting step see only the data objects of active agents and act on them independently from each other in parallel manner.
The program \( P_{\text{com}} \) performing the communication step sees all agents (both active and passive agents) and data objects of all agents.

A computation step of the system \( \Gamma \) is a state transition occurring either as the result of a rewriting step or as the result of a communication step. A computation performed by the system \( \Gamma \) is an alternating sequence of rewriting and communication steps. That is, a computation of the system \( \Gamma \) looks like this:

\[
S_0 \Rightarrow_{\text{rew}} S_2 \Rightarrow_{\text{comm}} S_3 \ldots S_{n-1} \Rightarrow_{\text{rew}} S_n
\]

The model introduced above indeed gives a uniform description of NLP systems. Its power will be demonstrated in the next section by translating PCLP and CCNLP models to this formalism.

5.3 NLP systems as ANRS systems

The aim of this section is to present possible representations of the models of networks of language processors described in Chapter 3 as ANRS systems. Subsection 5.3.1 introduces some notations and ext-HLRP programs which will be used for defining the ANRS systems corresponding to the models from Chapter 3. Subsection 5.3.2 deals with ext-HLRPs which simulate the effect of language processors. As particular examples, programs simulating Chomsky-grammars and splicing schemes will be presented in Subsection 5.3.2. Subsection 5.3.3 presents ANRS systems equivalent to CCNLP systems. General CCNLP systems, test-tube systems and CCPC grammars are rewritten to ANRS systems there. Subsection 5.3.4 presents representation of PCLP systems as ANRS systems. As a particular example, ANRS systems equivalent to PC grammar systems communicating by request are described there.

5.3.1 Preliminary preparations

The aim of this subsection is to present some programs and special categories which will be used in description of CCNLP and PCLP systems. These constructions are common for both CCNLP and PCLP systems, therefore it is reasonable to introduce them in a separate unit.

Strings over an alphabet \( V \) will be represented as directed labeled graphs with node labels from \( V \). That is, a string \( a_1 a_2 \ldots a_n \) will be represented as labeled graph

\[
\bullet \rightarrow \bullet \rightarrow \ldots \rightarrow \bullet
\]

Denote the labeled graph representation of a string \( \omega \) by \( \mathcal{M} \). The empty string \( \epsilon \) is simply the empty graph, which will be denoted by \( \emptyset \). The graph representation of a string consisting of a single symbol \( a \) will be denoted simply by \( a \). Generally, in case of labeled graphs, a node labeled with symbol \( l \) will be written simply as \( l \).
5.3. NLP SYSTEMS AS ANRS SYSTEMS

We will need to represent sets (or multisets) of strings. For our purposes, the following construction might be useful. Take a discrete graph \( Lab \), a graph \( LG \) such that \( Lab \subseteq LG \) and construct the category \( N_{D,LG,Lab} \). Elements of \( N_{D,LG,Lab} \) are of the form \( S = (G, m : G \rightarrow LG, f : V_{m^{-1}(Lab)} \rightarrow \text{Obj}(D)) \) where \( m^{-1}(Lab) \) is a discrete graph. Observe that \( S \) can be viewed as a multiset of objects of \( D \), where each element of the multiset is labeled by a symbol from \( Lab \). So for representing multisets of strings we will use objects of the category \( N_{LGraph, LG, Lab} \) where \( LGraph \) stands for the category of labeled graphs. Sets of strings will be represented by the same construction as multisets of strings.

In the subsequent text we assume that all categories of type \( N_{D,TG,Ag} \) have been constructed according to Theorem 5.2.2. Note that the category \( LGraph \) together with class of injective label preserving graph morphisms satisfies HLRP conditions. We will mainly use categories of the form \( N_{D,TG,Ag} \) where \( D = N_{LGraph, LG, Lab} \). To make defining ext-HLRP programs easier, the following abbreviation will be used. Consider the category \( N_{D,TG,Ag} \) and let \( K = (KG, m : KG \rightarrow TG) \) be a typed graph over \( TG \) such that

\[
|m^{-1}(Ag)| = 1
\]  

that is, it contains only one node labeled by a symbol from \( Ag \). Then each object of the form \( (KG, m, f) \) might be given by the following vector-like notation:

\[
\begin{bmatrix}
K \\
\text{f}(m^{-1}(Ag))
\end{bmatrix}
\]

Let’s denote by + the disjoint union of graphs. If \( m : G \rightarrow TG \) and \( l : H \rightarrow TG \) then \( m + l : G + H \rightarrow TG \) defined as \( m + l(x) = m(x) \) if \( x \in V_G \cup E_G \) and \( m + l(x) = l(x) \) otherwise. We can extend this notation to arbitrary \( 2 \leq n \). If we have an object \( Y = (KG_1 + KG_2 + \ldots + KG_n, m_1 + m_2 + \ldots + m_n, f) \) such that \( m_i : KG_i \rightarrow TG \) and \( |V_{m_i^{-1}(Ag)}| = 1 \) for each \( i = 1 \ldots n \) then the object \( Y \) will be denoted by

\[
\begin{bmatrix}
KG_1 & KG_2 & \ldots & KG_n \\
\text{f}(m_1^{-1}(Ag)) & \text{f}(m_2^{-1}(Ag)) & \ldots & \text{f}(m_n^{-1}(Ag))
\end{bmatrix}
\]  

(5.2)

If \( D \) itself is an ANRS category, then brackets around objects of \( D \) might be omitted in the notation above. That is, if \( D = N_{D', TG', Ag'} \) and for each \( i \) object \( \text{f}(m_i^{-1}(Ag)) = (DG_i, m_i, f_i) \) has the property that \( |V_{m_i^{-1}(Ag_i)}| = 1 \) then the object from (5.2) will look like this:

\[
\begin{bmatrix}
KG_1 & KG_2 & \ldots & KG_n \\
DG_1 & DG_2 & \ldots & DG_n \\
\text{f}_1(m_1^{-1}(Ag')) & \text{f}_2(m_2^{-1}(Ag')) & \ldots & \text{f}_n(m_n^{-1}(Ag'))
\end{bmatrix}
\]

(5.3)

If the graph \( DG_i \) is the empty graph for some \( i \) then the notation above will still be used. If \( DG_{i_k} \) where \( 1 \leq i_k \leq n \) and \( k \leq n \) are empty graphs and \( DG_j \) are...
non-empty graphs with the property that $|\tilde{m}^{-1}_j(Ag')| = 1$ ($j \notin \{i_1, i_2, \ldots, i_k\}$) then object (5.3) will be represented like this:

$$
\begin{bmatrix}
KG_1 & KG_2 & \cdots & KG_n \\
DG_1 & DG_2 & \cdots & DG_n \\
x_1 & x_2 & \cdots & x_n
\end{bmatrix}
$$

(5.4)

where $DG_{i_k} = \emptyset$ and $x_{i_k} = \emptyset$ and $x_j = \tilde{f}_j(\tilde{m}^{-1}_j(Ag'))$. For example, if $n = 2$ and $DG_1$ is an empty graph but $DG_2$ is not an empty graph then the object (5.3) will be denoted as

$$
\begin{bmatrix}
KG_1 & KG_2 \\
\emptyset & DG_1 \\
\emptyset & \tilde{f}_2(\tilde{m}^{-1}_2(Ag))
\end{bmatrix}.
$$

Notice that the representation above doesn’t contain the typing information about communication graphs. On the other hand for practical applications the nodes and the edges of communication graphs will be given together with their labels, so in this way the matrix-like representation won’t be ambiguous.

To make the notation above more clear, a small example will be presented.

**Example 5.3.1.** Denote by $K_L$ the graph $K_L = (L, L \times L, t, s)$ such that $t((a, b)) = b$ and $s((a, b)) = a$ for each $(a, b) \in L \times L$. That is, $K_L$ is the complete graph with set of nodes $L$.

Consider the category $N_{D,TG,Ag}$ where $D = N_{Graph,LG,LAg}$ Let $LG$ be the graph $K_{\{a,b,c\}}$. Let $LAg$ be the discrete graph $(\{a\}, \emptyset, \emptyset, \emptyset)$.

Let $TG$ be the graph $K_{\{A,B,C,D\}}$ and $Ag$ be the discrete graph $(\{A, B\}, \emptyset, \emptyset, \emptyset)$

Then consider the following hierarchical graph:
In the diagram above the arrows labeled by $H$ indicate the hierarchical relationship; the agent at the source of the arrow owns the object at the target of the arrow. Nodes are denoted by dots and labels of nodes are denoted by putting the label symbol of a node above the dot representing the node. That is, the picture above describes the object $O = (G_O, m_O, f_O)$ where $(G_O, m_O)$ is the following graph

![Graph Diagram]

and $f_O(A) = Z$ and $f_O(B) = Y$ where $Z = (G_Z, m_Z, f_Z)$ and $Y = (G_Y, m_Y, f_Y)$ such that $(G_Z, m_Z)$ is the typed graph:

![Typed Graph Diagram]

and $(G_Y, m_Y)$ is the typed graph

![Typed Graph Diagram]

Here $f_Z(a)$ is the following labeled graph

![Labeled Graph Diagram]

and $f_Y(a)$ is the following labeled graph

![Labeled Graph Diagram]

Then a valid matrix-like representation of the object $O$ would look like this:

$$
\begin{bmatrix}
A & C & D & B & C \\
b & a & b & a & a \\
R & W & x & x & y
\end{bmatrix}
$$

In the following text some programs important for further discussion will be presented.

To start with, language recognizing ext-HLPRs will be presented. Let $L$ be a recursive language (a recursive set of strings). Consider the category $N_{\text{Graph}}_{LG,Lab}$. Let $x, y, z$ be graphs typed over $LG$ with property (5.1), That is, each of graphs $x$, $y$ and $z$ contains only one node labeled by an agent type. Denote by $P_{L}^{x,y,z}$ the program acting on objects of $N_{\text{Graph}}_{LG,Lab}$ which, when applied to an object $S$, does the following: for a string which belongs to the object $S$ (recall that $S$ can be viewed as a multiset of strings such that each string is assigned to a node labeled by a symbol from $Lab$) and is assigned to the only node of $x$ labeled by a symbol from $Lab$ the program decides whether the string is in $L$ or not. That is, program $P_{L}^{x,y,z}$ grabs a subobject of the form $[x\ [w]$ ($w \in V^*$ ). In the former case (i.e. $w \in L$ ) the object $S$ is changed in that way that the subobject with communication graph $x$ (i.e. $[x\ [w]$) becomes
subobject with communication graph $y$ and the rest of the object remains unchanged. The string belonging to the only agent labeled node of $y$ will be the same as the string belonging to the only agent labeled node of $x$. If the string belonging to the agent labeled node of $x$ doesn’t belong to $L$ then the same $x$ is changed to $z$ in the similar way as described above. To illustrate it, if $\omega \in L$ then

$$
\left[ x \at \omega \right] \Rightarrow \mathcal{P}_{L}^{x,y,z} \left[ y \at \omega \right]
$$

else

$$
\left[ x \at \omega \right] \Rightarrow \mathcal{P}_{L}^{x,y,z} \left[ z \at \omega \right]
$$

The concrete shape of the program $\mathcal{P}_{L}^{x,y,z}$ depends of the language $L$, therefore it won’t be given here.

Another important class of programs is the program equivalent to branching constructions of imperative programming languages. Let $\mathcal{N}_{D,TG,A_{g}}$ be a ANRS state category, let $K = \{k_{1}, k_{2}, \ldots, k_{n}\}$ be a finite set of objects of $\mathcal{N}_{D,TG,A_{g}}$. Assume that $TG$ has nodes 1 and 0 and assume that the graph component of those objects of $\mathcal{N}_{D,TG,A_{g}}$ which we are interested in does not contain nodes labeled by 1 or 0. Let $S_{1}, S_{2}$ be ext-HLRP programs over $\mathcal{N}_{D,TG,A_{g}}$. Then program CHECK($K,S_{1},S_{2}$) implements a conditional branching: when applied to an object $G$ it checks whether any of elements of $K$ occurs in $G$ and if so it applies $S_{1}$ to $G$, otherwise it applies $S_{2}$ to $G$. A possible implementation of this program is the following

$$
CHECK(K,S_{1},S_{2}) = \{(\text{empty} \leftarrow \text{empty} \rightarrow \text{ch}(0))\};
$$

$$
\{p_{k_{1}}, \ldots, p_{k_{n}}\} \downarrow;
$$

$Start(S_{1},1); Start(S_{2},0)$

$p_{k_{i}} = (k_{i}(0) \leftarrow k_{i} \rightarrow k_{i}(1))$

where

$k_{i} = (G,m,f)$ and $k_{i}(z) = (G_{z},m_{z},f)$

where

$G = (V_{G}, E_{G}, t_{G}, s_{G})$ and

$G_{z} = (V_{G} \cup \{1\}, E_{G}, t_{G}, s_{G})$, $m_{z}(x) = \begin{cases} m(x) & \text{if } x \in V_{G} \cup E_{G} \\ 1 & \text{if } x = z \end{cases}$ for each $z \in \{0,1\}$

$Start(S,z) = ((\text{ch}(z) \leftarrow \text{empty} \rightarrow \text{empty})\}; S) \downarrow$
where 

\[ ch(z) = ((\{z\}, \emptyset, \emptyset), id_1, \emptyset) \]

and 

\[ empty = (\emptyset, \emptyset, \emptyset) \]

The program \( CHECK(K, S_1, S_2) \) has been introduced in [14] for labeled graphs. The implementation given above is a generalization of the solution given there. The implementation works as follows. When applied to an object \( G \), a node with label 0 is inserted to the communication graph of the object \( G \). Then one of the productions \( p_k \) is applied to \( G \). If there exists a production \( p_k \) applicable to \( G \) then it means that \( G \) contains a subobject \( k \). In this case the newly inserted node with label 0 is deleted and a node with label 1 is added. Program \( Start(S, z) \) leaves an object \( H \) intact, if it doesn’t contain a node labeled by \( z \). Otherwise it deletes the node labeled by \( z \) and applies the program \( S \) to the rest of the object \( H \). That is, \( Start(S_1, 1) \) executes \( S_1 \) if and only if a node with label 1 has been introduced, that is, a \( k \in K \) occurs in the object \( G \). \( Start(S_2, 0) \) executes \( S_2 \) if and only if a node labeled by 0 is present, that is, no production \( p_k \) could be applied. That means that no \( k \in K \) occurs in the object \( G \). That is, the implementation above indeed fits the specification of the program.

Consider again the category \( CAT = N_{D, TG, Ag} \) where \( D = N_{L, Graph, LG, Lib} \). Let be \( a, b \in V_A \) and \( x, y \) graphs typed over \( LG \) with the property (5.1) The ext-HLRP \( COPY^{(a, x), (b, y)} \) over \( CAT \) does the following. When applied to an object of the form \((G, m, f)\) it chooses a subobject of the form \( \begin{bmatrix} a \\ x \\ \overline{w} \end{bmatrix} (w \in V^*) \).

If there exists a subobject of the form \( \begin{bmatrix} b \\ \emptyset \\ \overline{w} \end{bmatrix} \) then the program transforms this subobject to the subobject \( \begin{bmatrix} b \\ y \\ \overline{w} \end{bmatrix} \). Otherwise, \( COPY^{(a, x), (b, y)} \) can’t be applied to object \((G, m, f)\). That is, the effect of \( COPY^{(a, x), (b, y)} \) can be described in the following way:

\[
\begin{bmatrix} a \\ x \\ \overline{w} \\ \emptyset \end{bmatrix} \Rightarrow COPY^{(a, x), (b, y)} \begin{bmatrix} a \\ x \\ \overline{w} \\ y \end{bmatrix}
\]

Program \( MOVE^{(a, x), (b, y)} \) does the same as \( COPY^{(a, x), (b, y)} \) except that it removes the source string together with graph \( x \). That is, its effect can be described in the following way:

\[
\begin{bmatrix} a \\ x \\ \overline{w} \\ \emptyset \end{bmatrix} \Rightarrow MOVE^{(a, x), (b, y)} \begin{bmatrix} a \\ \emptyset \\ y \\ \overline{w} \end{bmatrix}
\]

One of the possible implementations of \( COPY^{(a, x), (b, y)} \) might look like this:
COPY\(^{(a,x),(b,y)}\) = INIT; COPY \downarrow; FIN

\[
INIT = \text{CHECK}\left\{ \begin{array}{c}
\begin{bmatrix}
  a \\
  x \\
  \alpha
\end{bmatrix} \quad | \quad \alpha \in V, \{p_{1,\alpha,\beta}|\alpha \in V, \beta \in V \cup \{\epsilon\}\}, \{p_{\text{empty}}\} \\
\end{array}\right\}
\]

\[
p_{1,\alpha,\beta} = \begin{bmatrix}
  a \\
  x \\
  \alpha \rightarrow \beta \\
\end{bmatrix}\quad \leftrightarrow\quad \begin{bmatrix}
  a \\
  x \\
  \beta
\end{bmatrix} \quad \rightarrow\quad \begin{bmatrix}
  a \\
  x \\
  R \rightarrow \alpha \rightarrow \beta \\
\end{bmatrix}
\]

\[
p_{1,\alpha,\epsilon} = \begin{bmatrix}
  a \\
  x \\
  \alpha
\end{bmatrix}\quad \leftrightarrow\quad \begin{bmatrix}
  a \\
  x \\
  \beta
\end{bmatrix} \quad \rightarrow\quad \begin{bmatrix}
  a \\
  x \\
  R \rightarrow \alpha \\
\end{bmatrix}
\]

\[
p_{\text{empty}} = \begin{bmatrix}
  b \\
  \emptyset \\
\end{bmatrix}\quad \leftrightarrow\quad \begin{bmatrix}
  b \\
  \emptyset \\
\end{bmatrix} \quad \rightarrow\quad \begin{bmatrix}
  b \\
  \emptyset
\end{bmatrix}
\]

\[
\text{COPY} = \{p_{\alpha,\beta}|\alpha, \beta \in V\}
\]

\[
p_{\alpha,\beta} = \begin{bmatrix}
  a \\
  x \\
  R \rightarrow \alpha \rightarrow \beta \\
\end{bmatrix}\quad \leftrightarrow\quad \begin{bmatrix}
  a \\
  x \\
  \alpha \rightarrow \beta \\
\end{bmatrix} \quad \rightarrow\quad \begin{bmatrix}
  a \\
  x \\
  \alpha \rightarrow \beta \leftarrow R \\
\end{bmatrix}
\]

\[
\text{FIN} = \{\text{del}_\alpha|\alpha \in V\} \downarrow
\]

\[
\text{del}_\alpha = \begin{bmatrix}
  a \\
  x \\
  R \rightarrow \alpha \quad W \rightarrow \alpha
\end{bmatrix}\quad \leftrightarrow\quad \begin{bmatrix}
  a \\
  x \\
  \alpha
\end{bmatrix} \quad \rightarrow\quad \begin{bmatrix}
  a \\
  x \\
  \alpha
\end{bmatrix}
\]

Program INIT copies the first symbol of the string of a node labeled by \(a\) to a node labeled by \(b\) and marks the first symbol of the node labeled by \(a\) by \(R\) and marks the symbol which has been copied to the node with label \(b\) by \(W\). Nodes \(R\) and \(W\) are used as reading and writing heads. The construction of INIT ensures that only the first symbol of the string of the node labeled by \(a\) gets marked by \(R\). During copying always the symbol standing after the symbol marked with \(R\) is copied and it is placed after the symbol marked with \(W\). Then nodes \(R\) and \(W\) are moved appropriately too. This is done by program COPY. Program FIN removes the nodes \(R\) and \(W\).

Program MOVE\(^{(a,x),(b,y)}\) can be implemented in similar manner.

MOVE\(^{(a,x),(b,y)}\) = INIT; MOVE \downarrow; FIN

\[
INIT = \text{CHECK}\left\{ \begin{array}{c}
\begin{bmatrix}
  a \\
  x \\
  \alpha
\end{bmatrix} \quad | \quad \alpha \in V, \{p_{1,\alpha,\beta}|\alpha \in V, \beta \in V \cup \{\epsilon\}\}, \{p_{\text{empty}}\} \\
\end{array}\right\}
\]
\[ p_{1, \alpha, \beta} = \left( \begin{array}{c|c} \alpha \to \beta & 0 \\ \hline x & 0 \end{array} \right) \left( \begin{array}{c|c} \alpha \to \beta & 0 \\ \hline x & 0 \end{array} \right) \to \left( \begin{array}{c|c} \alpha \to \beta & 0 \\ \hline R \to \alpha \to \beta & 0 \end{array} \right) \] 

\[ p_{1, \alpha, \epsilon} = \left( \begin{array}{c|c} \alpha \to 0 & 0 \\ \hline x & 0 \end{array} \right) \left( \begin{array}{c|c} \alpha \to 0 & 0 \\ \hline x & 0 \end{array} \right) \to \left( \begin{array}{c|c} \alpha \to 0 & 0 \\ \hline R \to \alpha & 0 \end{array} \right) \] 

\[ p_{\text{empty}} = \left( \begin{array}{c|c} b & 0 \\ \hline \emptyset & 0 \end{array} \right) \left( \begin{array}{c|c} b & 0 \\ \hline \emptyset & 0 \end{array} \right) \to \left( \begin{array}{c|c} b & 0 \\ \hline \emptyset & 0 \end{array} \right) \] 

\[ \text{MOVE} = \{ p_{\alpha, \beta, \gamma} | \alpha, \beta \in V \} \] 

\[ p_{\alpha, \beta} = \left( \begin{array}{c|c} a & b \\ \hline x & 0 \end{array} \right) \left( \begin{array}{c|c} a & b \\ \hline x & 0 \end{array} \right) \to \left( \begin{array}{c|c} a & b \\ \hline x \to \beta & 0 \end{array} \right) \] 

\[ \text{FIN} = \{ \text{del}_{\alpha} | \alpha \in V \} \downarrow \] 

\[ \text{del}_{\alpha} = \left( \begin{array}{c|c} a & b \\ \hline R \to \alpha & 0 \end{array} \right) \left( \begin{array}{c|c} a & b \\ \hline R \to \alpha & 0 \end{array} \right) \to \left( \begin{array}{c|c} a & b \\ \hline \emptyset \to \beta & 0 \end{array} \right) \] 

If we omit \( a \) and \( b \) from productions of the programs defined above we get programs \( \text{COPY}^{x, y} \) and \( \text{MOVE}^{x, y} \) \( (x, y \in \text{Lab}) \) with essentially the same effect.

Again consider the category \( N_{D, T_G, A_g} \) where \( D = N_{\text{Graph}, L_G, \text{Lab}} \). Let \( x \) and \( y \) be graphs typed over \( LG \) satisfying condition (5.1). Let \( a, b \in V_{A_g} \) be agent types. The program \( \text{APP}^{\text{COPY}}_{(a, x), (b, y)} \) appends the string with label in \( x \) of the agent \( a \) to a string with label in \( y \) of the agent \( b \). That is, when applied to an object \( G \), it grabs a subobject of the form \( \left[ \begin{array}{c} a \\ x \end{array} \right] \) \( (u \in V^*) \). If there is no such a subobject then \( \text{APP}^{\text{COPY}}_{(a, x), (b, y)} \) can’t be applied to \( G \). Then it looks for a subobject of the form \( \left[ \begin{array}{c} b \\ y \end{array} \right] \) \( (v \in V^*) \) and transforms this subobject to the subobject of the form \( \left[ \begin{array}{c} y \\ v \end{array} \right] \). That is, the following holds:

\[
\left[ \begin{array}{c} a & b \\ x & y \end{array} \right] \Rightarrow_{\text{APP}^{\text{COPY}}_{(a, x), (b, y)}} \left[ \begin{array}{c} a & b \\ u & v \end{array} \right]
\]
where $u,v \in V^*$. Like for $\text{COPY}^{x,y}$ it is also possible to define the program $\text{APP}^{\text{COPY}}_{x,y}$ which arises from $\text{APP}^{\text{COPY}}_{(a,x),(b,y)}$ by omitting $a$ and $b$ from the rules constituting the program. The following program is a possible implementation of $\text{APP}^{\text{COPY}}_{(a,x),(b,y)}$.

\[\begin{align*}
\text{START} &= \{p_{a,x,l,a,\beta,R}, p_{a,x,a,R} | \alpha \in V, \beta \in V\}; \\
&\{p_{b,y,r,a,\alpha,W} | \alpha, \beta \in V\}; \\
&\{fst_{\alpha,\beta} | \alpha, \beta \in V\}
\end{align*}\]

\[\begin{align*}
p_{z,k,r,a,\beta,T} &= \left( \begin{array}{c}
\left[ \begin{array}{c}
z \\
k
\end{array} \right] \\
\alpha \rightarrow \beta
\end{array} \right) \rightarrow \left( \begin{array}{c}
\left[ \begin{array}{c}
z \\
k
\end{array} \right] \\
\beta \\
T \rightarrow \alpha \rightarrow \beta
\end{array} \right)
\end{align*}\]

\[\begin{align*}
p_{z,k,a,T} &= \left( \begin{array}{c}
\left[ \begin{array}{c}
z \\
k
\end{array} \right] \\
\alpha \\
T \rightarrow \alpha
\end{array} \right)
\end{align*}\]

\[\begin{align*}
p_{z,k,l,a,\beta,T} &= \left( \begin{array}{c}
\left[ \begin{array}{c}
z \\
k
\end{array} \right] \\
\alpha \leftarrow \beta \\
T \rightarrow \alpha \leftarrow \beta
\end{array} \right)
\end{align*}\]

\[\begin{align*}
&\text{fst}_{\alpha,\beta} = \\
&\left( \begin{array}{c}
\left[ \begin{array}{c}
a \\
x
\end{array} \right] \\
\alpha \leftarrow R \\
\beta \\
W
\end{array} \right) \rightarrow \\
&\left( \begin{array}{c}
\left[ \begin{array}{c}
a \\
x
\end{array} \right] \\
\alpha \leftarrow R \\
\beta \\
\rightarrow \alpha \leftarrow \beta
\end{array} \right)
\end{align*}\]

\[\begin{align*}
\text{APP}^{\text{COPY}}_{(a,x),(b,y)} &= \text{CHECK}(\{b
\left[ \begin{array}{c}
y \\
\beta
\end{array} \right] | \beta \in V\}, \text{START}; \text{COPY} \downarrow; \text{FIN}, \text{COPY}^{(a,x),(b,y)})
\end{align*}\]

In the description above programs \text{FIN} and \text{COPY} are the same as for \text{COPY}^{(a,x),(b,y)}. The role of nodes labeled by $R$ and $W$ is the same as in the case of \text{COPY}^{(a,x),(b,y)}. Program \text{START} is responsible for placing nodes $R$ and $W$ correctly.

Consider again a category of the form $D = N_{\text{Graph}}, TG, Ag$. Let $x,y$ be graphs typed over $TG$ such that $x$ and $y$ satisfy condition (5.1). Let be $\alpha \in V$ Program $\text{COPY}^{x,y,\alpha}$ replaces the occurrence of symbol $\alpha$ in the string with label in $y$ with a copy of the string labeled by the only agent node of $x$. That is, it grabs a subobject of the form $\left[ \begin{array}{c}
x \\
w\end{array} \right] (w \in V^*)$. Then it grabs a subobject of the form $\left[ \begin{array}{c}
y \\
\alpha t
\end{array} \right] (s,t \in V^*)$ and changes it to the subobject $\left[ \begin{array}{c}
y \\
\alpha s t
\end{array} \right]$. That is,
its effect can be illustrated by the following example

\[
\begin{bmatrix}
  x \\
  y \\
  s\alpha t
\end{bmatrix} \quad \Rightarrow \quad \text{COPY}^{x,y,a}
\begin{bmatrix}
  x \\
  y \\
  s\alpha t
\end{bmatrix}
\]

The implementation of this program might be easily derived from that of \( \text{COPY}^{x,y} \), so it will be omitted here.

Program \( \text{SKIP} \) is the programs which doesn’t do anything:

\[
\text{SKIP} = \{(\emptyset \leftarrow \emptyset \rightarrow \emptyset)\}
\]  

Consider the category \( N_{L\text{Graph}_{TG,Ag}} \). Let be \( K \) a directed graph typed over \( TG \) (that is \( K = (KG, m : KG \rightarrow TG) \) such that \( |m^{-1}(Ag)| = 1 \). Assume that we are interested only in objects whose agents contain graphs labeled over node label alphabet \( Lab_k \) and edge label alphabet \( Lab_e \). Assume that the communication graph of objects of our interest do not contain nodes labeled by 1. Let \( DEL_K \) be an ext-HLRP over \( N_{L\text{Graph}_{TG,Ag}} \) such that it deletes those subobjects of objects, whose communication graphs coincide with \( K \).

\[
DEL_K = \text{MARK}; DEL \downarrow; FIN
\]

\[
\text{MARK} = \{(\begin{bmatrix}
  KG' \\
  \emptyset
\end{bmatrix} \leftarrow \begin{bmatrix}
  KG \\
  \emptyset
\end{bmatrix} \rightarrow \begin{bmatrix}
  KG'
\end{bmatrix})\}
\]

where \( KG' = (V_{KG} \cup \{1\}, E_{KG} \cup \{1_e\}, t, s) \)

and \( t(e) = \begin{cases} t(e) & e \in E_{KG} \\ m^{-1}(Ag) & e = 1_e \end{cases} \)

and \( s(e) = \begin{cases} s(e) & e \in E_{KG} \\ 1 & e = 1_e \end{cases} \)

\[
DEL = \{\text{del}_{a,b} \mid a, b \in Lab_k \wedge e \in Lab_e\}
\]

\[
\text{del}_{a,b} = (\begin{bmatrix}
  KG' \\
  a \rightarrow b
\end{bmatrix} \leftarrow \begin{bmatrix}
  KG'
\end{bmatrix} \rightarrow \begin{bmatrix}
  KG'
\end{bmatrix})
\]

\[
\text{del}_a = (\begin{bmatrix}
  KG'
\end{bmatrix} \leftarrow \begin{bmatrix}
  KG'
\end{bmatrix} \rightarrow \begin{bmatrix}
  KG'
\end{bmatrix})
\]

\[
FIN = \{\begin{bmatrix}
  KG'
\end{bmatrix} \leftarrow \emptyset \rightarrow \emptyset\}
\]
5.3.2 Language processors as ext-HLRP

As it was shown in [14] even programmed graph transformation systems are computationally universal, that is, they are able to simulate Turing-machines. Since ext-HLRP systems are extensions of programmed graph transformation systems they are also Turing equivalent over category of labeled graphs. From this it follows that rewriting of strings performed by language processors should be easily simulated by extended high-level replacement systems over a suitable category. To demonstrate this, simulation of Chomsky-grammars and splicing schemes will be given.

First, the ext-HLRPs simulating Chomsky-grammars will be presented. Let $G = (N,T,S,P)$ be a Chomsky-grammar. Let $p \rightarrow q \in P$ be a rewrite rule of $G$. Let $\overline{p},\overline{q}$ labeled graph representations of strings $p, q$ respectively. The following ext-HLRP $P_{p \rightarrow q}$ over $\text{LGraph}$ simulates the application of $p \rightarrow q$ to strings:

$$P_{p \rightarrow q} = \{p_a,b,p_a,p_b | a,b \in V\}$$

$$p_{a,b} = ((a \rightarrow p \rightarrow b) \leftarrow (a b) \rightarrow (a \rightarrow p \rightarrow b))$$

$$p_a = ((a \rightarrow p) \leftarrow (a) \rightarrow (a \rightarrow p))$$

$$p_b = ((\overline{p} \rightarrow \overline{b}) \leftarrow (\overline{b}) \rightarrow (\overline{p} \rightarrow \overline{b}))$$

Then let $AP_G = \bigcirc_{p \rightarrow q} P_{p \rightarrow q}$. Then we have that

$$x \Longrightarrow p \iff x \Longrightarrow AP_G \overline{p}$$

which implies that

$$x \in L(G) \iff x \Longrightarrow AP_G \overline{x}$$

That is, $AP_G = AP_G \downarrow$ simulates the behaviour of Chomsky-grammar $G$.

Now the ext-HLRP simulating the work of an arbitrary splicing scheme will be presented. The program simulating splicing schemes will work on sets (multisets of strings), that is, on objects of some category $N_{\text{LGraph},L,G,Lab\cup\{1,2\},l\in\text{Lab}}$. Labels $1_l,2_l$ are supposed to be used during the simulation of splicing operation. It will be assumed the objects to which the program is applied have no nodes labeled by $1_l$ or $2_l$ ($l \in \text{Lab}$). Let $r = p\#q\#u\#v$ be a splicing rule. Then consider the following program:

$$SP_r = \bigcirc_{l \in \text{Lab}} \text{COPY}^{r,1_l}, \bigcirc_{l \in \text{Lab}} \text{COPY}^{r,2_l}, \text{SPLICING}_r; \bigcirc_{l \in \text{Lab}} \text{MOVE}^{1_l,l}; \bigcirc_{l \in \text{Lab}} \text{MOVE}^{2_l,l}$$
5.3. NLP SYSTEMS AS ANRS SYSTEMS

The program $SP_r$ picks up two strings with labels $l$ and $l'$ and makes a copy of them with labels $1_l$ and $2_l$. Then the program $SPLICING_r$ executes the splicing operation on strings with labels $1_l$ and $2_l$, in such a way that it looks for the substring $pq$ in the string with label $1_l$ and it looks for the substring $uv$ in the string labeled by $2_l$. If it finds those substrings then it executes the splicing operation. If we assume that the string with label $1_l$ is of the form $x_1pqy$ and the string with label $2_l$ is of the form $swt$ then after applying $SPLICING_r$ to them we will get a string $x_1pqt$ with label $1_l$ and a string $swt_2y$ with label $2_l$. Finally the string with label $1_l$ gets label $l$ and the string with label $2_l$ gets label $l'$. One of possible implementations of $SPLICING_r$ is the following:

$$p_{k,b,d} = \left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$p_{k,b,e} = \left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$p_{k,c,d} = \left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$p_{k,e,c} = \left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
1_l \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{b \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
2_k \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}} \\
\overrightarrow{\mathcal{P}}
\end{array} \right)_{d \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$p_{k,l,k,b,a} = \left( \begin{array}{c}
1_l \\
R_1 \\
R_2 \\
R_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
W_2 \\
W_1 \\
W_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
1_l \\
R_1 \\
R_2 \\
R_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
W_2 \\
W_1 \\
W_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$p_{k,l,k,a,b} = \left( \begin{array}{c}
1_l \\
R_1 \\
R_2 \\
R_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
W_2 \\
W_1 \\
W_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
1_l \\
R_1 \\
R_2 \\
R_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
W_2 \\
W_1 \\
W_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$p_{fin,k,l,1,a} = \left( \begin{array}{c}
1_l \\
R_1 \\
R_2 \\
R_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
W_2 \\
W_1 \\
W_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
1_l \\
R_1 \\
R_2 \\
R_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
2_k \\
W_2 \\
W_1 \\
W_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$p_{fin,k,l,2,a} = \left( \begin{array}{c}
2_k \\
R_2 \\
R_2 \\
R_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
1_k \\
W_1 \\
W_1 \\
W_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
\left( \begin{array}{c}
2_k \\
R_2 \\
R_2 \\
R_2
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}} \left( \begin{array}{c}
1_k \\
W_1 \\
W_1 \\
W_1
\end{array} \right)_{a \rightarrow \overrightarrow{\mathcal{P}}}
$$

$$SPLICING_r = \{ p_{k,l,b,d} | k, l \in Lab \land b, d \in V \cup \{ \epsilon \} \};$$

$$\{ p_{k,l,k,a,b} | 1, 2 \land a, b \in V \land l, k \in Lab \} \downarrow;$$

$$\{ p_{fin,k,l,i,a} | a \in V \land i = 1, 2 \land l, k \in Lab \} \downarrow;$$
Then, given a splicing scheme \( \sigma = (V, P) \) the effect of applying \( \sigma \) to a set of strings \( L \) corresponds to the effect of applying \( SP_\sigma \) to the object \( (G, m, f) \) such that \( L = \bigcup_{v \in V_\ell} f(v) \). More precisely: 
\[
(G, m, f) \implies_{SP_\sigma} (D, m', \bar{f}) \implies \sigma^*(L) = \bigcup_{v \in V_\ell} \bar{f}(v)
\]

5.3.3 CCNL systems

In this subsection representation of CCNL systems in terms of ANRS systems will be presented. First, the representation of the general model will be given, then two particular cases, namely CCPC grammar systems and test-tube systems, will be rewritten to ANRS systems.

Relying on Subsection 5.3.2 we will assume that each language processor \( \Pi = (V, M, P) \) working in mode \( f \) can be simulated by the ext-HLRP \( P_\Pi \) in the following way. If we take an object \( Z = (G, m, f) \) of \( N_{\text{Graph}, LG, Lab} \), and denote by \( \text{Lang}(Z) = \bigcup_{v \in V_{\ell-1(Lab)}} \{ f(v) \} \) the supporting set of \( Z \) (recall that \( Z \) can be viewed as a multiset) then

\[
\Pi' (\text{Lang}(Z)) = \text{Lang}(D) \text{ if and only if } Z \implies_{P_\Pi} D
\]

So in the general case we have just to deal with the communication. As the assumption concerning the simulation of language processors probably reveals, we will represent sets of strings by objects from \( N_{\text{Graph}, LG, Lab} \). The representation of CCNL systems goes as follows

**Definition 5.3.1.** Let \( \Gamma = (V, M, A_1, \ldots, A_n, R) \) be a CCNL system defined in Definition 3.2.1. Define the ANRS system corresponding to system \( \Gamma \) as follows:

\[
AN = (TG, A_g, D, S_0, A_{g_{\text{act}}}, P_{\text{comm}}, \{ P_a | a \in V_{A_{\text{act}}} \})
\]

where

- \( A_g \) is a discrete graph with set of vertices being equal to \( \{1, 2, \ldots, n\} \)
- \( TG \) is the type graph of communication graphs, equals to \( A_g \)
- \( A_{g_{\text{act}}} \) is the graph of active agents, equals to \( A_g \)
- \( D \) is the category of data objects. Define \( D \) to be the category \( N_{L, \text{Graph}, LG, Lab} \)

where

\[
\text{Lab} = \{ a, \bar{a} \}
\]

and

\[
LG = (V_{LG}, V_{LG} \times V_{LG}, pr_1, pr_2)
\]

where \( pr_l(l, k) = k \) and \( pr_2(l, k) = l \) for each \( (l, k) \in V_{LG} \times V_{LG} \) and

\[
V_{LG} = \text{Lab} \cup \{ \bar{a}, \bar{c}, o', o'' \} \cup \{ \bar{i} | i = 1 \ldots n \}
\]
5.3. NLP SYSTEMS AS ANRS SYSTEMS

$S_0$ is the initial configuration of the system, $S_0 = \begin{bmatrix} 1 & S_1 & 2 & \cdots & n \end{bmatrix}$ where

$S'_i = (G_i, m_i, f_i)$ such that

- $G_i$ is a discrete graph
- $m_i(v) = o$ for all $v \in V_{G_i}$
- $f_i : V_{G_i} \rightarrow \text{Obj}(LGraph)$ is injective
- $\text{Lang}(S'_i) = S_i$ where $S_i$ is the set of initial axioms of the $i$th component

$P_{\text{comm}}$ is the program encoding communication steps

$P_a$ simulates the work of the $i$th language processor $\Pi_i$ in a sense explained above

Then program $P_{\text{comm}}$ depends on the communication protocol used by the system. Recall that three communication protocols have been defined for CC-NLP systems. We shall give the program $P_{\text{comm}}$ for each of them

**protocol (a)**

$$P'_{ij} = H_iP_{O_i}^{i,o \rightarrow \sigma} \circ \Pi_j^{j,o \rightarrow \sigma}; \text{CHECK}\{\begin{bmatrix} i \\sigma \\emptyset \end{bmatrix}\}, \text{COPY}\{i,o \rightarrow \sigma, j, \bar{\sigma}\}; H_jP_{I_j}^{j,\bar{\sigma} \rightarrow o', \sigma \rightarrow o''}; \text{SKIP};$$

$$\text{CHECK}\{\begin{bmatrix} j \\emptyset \sigma \end{bmatrix}\}, \begin{bmatrix} i \\sigma \emptyset \end{bmatrix} \leftrightarrow \begin{bmatrix} i \\sigma' \emptyset \end{bmatrix} \leftrightarrow \begin{bmatrix} i \\sigma \emptyset \end{bmatrix}\}; \text{SKIP}$$

$$P_{ij} = \overline{P'_{ij}}$$

$$P_i = \overline{\bigcup_{k=1}^4 P_i^k}$$

$$P_i^1 = H_i\text{DEL}_{\sigma \rightarrow o''}$$

$$P_i^2 = \bigcup_{S \subseteq \{\bar{T}, \bar{\sigma}, \ldots, \bar{\pi}\}\backslash \{\bar{T}\}} H_i\text{DEL}_S$$

$$D_i^S = \text{DEL}_{G_S} \text{ for } S \subseteq \{\bar{T}, \bar{\sigma}, \ldots, \bar{\pi}\}\backslash \{\bar{T}\}$$

$$D_i^{\{\bar{T}, \bar{\sigma}, \ldots, \bar{\pi}\}\backslash \{\bar{T}\}} = \{\begin{bmatrix} G^{\{\bar{T}, \bar{\sigma}, \ldots, \bar{\pi}\}\backslash \{\bar{T}\}} \emptyset \end{bmatrix} \leftrightarrow \begin{bmatrix} o \emptyset \end{bmatrix} \rightarrow \begin{bmatrix} o' \emptyset \end{bmatrix}\}$$

where $\text{DEL}_S = (\{o\} \cup S, \{e_k | k \in S\}, t, s)$ such that $t(e_k) = o$ and $s(e_k) = k$

$$P_i^3 = \text{MOVE}\{i, \delta \rightarrow o', \sigma \}$$
\[ P_t^4 = H_t\{ \left[ o \rightarrow \overline{\sigma} \right] \left[ o \right] \rightarrow \left[ o \right] \} \]

\[ P_{comm} = \text{CHECK}\left\{ \left[ i \right], \left[ o \right] \mid 1 \leq i \leq n ; \right\}_{i,j=1}^{n} P_{ij}^{n} F_{i}, S_{KIP} \right\} \]

Program \( P_{ij} \) simulates the exchange of information between the \( i \)th and the \( j \)th agent. It executes instances of programs \( P_{ij}^n \) in parallel, one instance for each string in the repository of the \( i \)th agent. Program \( P_{ij}^n \) picks up a string from the string repository of the \( i \)th agent. It tests whether it passes the output filter of the \( i \)th agent. Labels of strings which have passed the output filter of the \( i \)th agent are marked with \( \overline{\sigma} \). Labels of strings which have not passed the output filter of the \( i \)th agent are marked with \( \overline{\sigma} \). Strings with labels marked with \( \overline{\sigma} \) are copied to the \( j \)th agent. The new copies have labels \( \overline{o} \). If the newly copied string doesn’t pass the input filter of the \( j \)th agent, then it is marked by \( o \) besides its source string in the \( i \)th agent repository is marked by \( \overline{j} \).

Program \( P_{ij}^n \) changes the label of those new strings which have passed the input filter of the \( i \)th agent (strings with label \( \overline{o} \leftarrow o \) ) to \( o \), deletes those new strings which haven't passed the input filter of the \( i \)th agent (strings with label \( \overline{o} \leftarrow o' \) ) and deletes all those strings which managed to pass its output filter and some other agent’s input filter, that is, those strings which are labeled by \( G_s \) for some \( S \subseteq \{1,2,\ldots,n\} \backslash \{i\} \). The last step is done by programs \( \text{DEL}_{G_s} \). Notice that if \( S \subseteq S' \) and \( \text{DEL}_{G_s(S')} \) can be executed, then \( \text{DEL}_{G_s(S')} \) can’t be executed, since there are edges between elements of \( S' \backslash S \) and the node labeled by \( o \).

Program \( P_{comm} \) checks whether there is any agent whose repository is not empty. If such an agent doesn’t exist, then \( P_{comm} \) doesn’t change the object it has been applied to.

**protocol (b)**

\[ P_{ij} = \left[ P_{ij}^n \right] \]

\[ P_{ij}^n = H_i P_{G_i}^{o \rightarrow \overline{\sigma}, o} ; \]

\[ \text{COP}Y(i,o \rightarrow \overline{\sigma},(j,\overline{o})) ; \]

\[ H_j P_{ij}^{\overline{o}, \overline{o} \leftarrow o} ; \]

\[ H_j \text{DEL}_{o \leftarrow o'} ; \]
53. NLP SYSTEMS AS ANRS SYSTEMS

\[
\{\left[ \begin{array}{c}
  i \\
  o \\
  \emptyset
\end{array} \right] \leftrightarrow \left[ \begin{array}{c}
  i \\
  o \\
  \emptyset
\end{array} \right] \rightarrow \left[ \begin{array}{c}
  i \\
  o \\
  \emptyset
\end{array} \right] \}
\]

\[
P'_{\text{comm}} = \prod_{i,j=1,(i,j) \in R} P_{ij}; C\text{HECK}\left(\left\{\left[ \begin{array}{c} i \\
  \hat{o} \\
  \emptyset\end{array} \right] \right\} | 1 \leq i \leq n\right), \prod_{i=1}^{n} \text{MOVE}(i, \hat{o}, (i, o), \text{SKIP})
\]

\[
P_{\text{comm}} = C\text{HECK}\left(\left\{\left[ \begin{array}{c} i \\
  o \\
  \emptyset\end{array} \right] \right\} | 1 \leq i \leq n\right), P'_{\text{comm}}, \text{SKIP}
\]

As in the case of protocol (a) \( P_{ij} \) simulates the interaction of the \( i \)th and the \( j \)th agent. \( P_{ij} \) marks strings which passes the \( i \)th agent’s output filter with label \( o \leftarrow \hat{o} \). Then it copies those strings to the \( j \)th agent. The \( j \)th agent selects one of those, which pass its input filter. Strings which have passed the \( j \)th agent’s input filter are marked by \( \hat{o} \), those which haven’t passed it are marked by \( \hat{o} \leftarrow o' \) and are deleted from the \( j \)th agent’s repository. Program \( \text{MOVE}(i, \hat{o}, (i, o)) \) changes the labels of strings labeled by \( \hat{o} \) to label \( o \).

**protocol (c)**

\[
P_{\text{comm}} = C\text{HECK}\left(\left\{\left[ \begin{array}{c} i \\
  o \\
  \emptyset\end{array} \right] \right\} | 1 \leq i \leq n\right), P'_{\text{comm}}, \text{SKIP}
\]

\[
P'_{\text{comm}} = \prod_{i,j=1,(i,j) \in R} P_{ij};
\]

\[
\prod_{i=1}^{n} H_{i} \text{DEL}_{o};
\]

\[
C\text{HECK}\left(\left\{\left[ \begin{array}{c} i \\
  \hat{o} \\
  \emptyset\end{array} \right] \right\} | 1 \leq i \leq n\right), \prod_{i=1}^{n} \text{MOVE}(i, \hat{o}, (i, o), \text{SKIP})
\]

Here \( P_{ij} \) exactly the same as for the protocol (b). We need \( H_{i} \text{DEL}_{o} \) to delete the original strings of the \( i \)th agents. Notice that according to the protocol (b) the original strings of the \( i \)th agent must remain in the repository of the \( i \)th agent.

It is easy to see, that \( P_{\text{comm}} \) simulates the rewriting step of CCNLP systems in the following sense:

\[
\left[ \begin{array}{cccc}
  1 & 2 & \ldots & n \\
  x_{1} & x_{2} & \ldots & x_{n}
\end{array} \right] \Rightarrow P_{\text{comm}} \left[ \begin{array}{cccc}
  1 & 2 & \ldots & n \\
  y_{1} & y_{2} & \ldots & y_{n}
\end{array} \right]
\]

if and only if

\[
(Lang(x_{1}), Lang(x_{2}), \ldots, Lang(x_{n})) \Rightarrow_{\text{comm}} (Lang(y_{1}), Lang(y_{2}), \ldots, Lang(y_{n}))
\]
Because of the assumption on the effect of the programs $P_i$ we have that

$$
\begin{bmatrix}
1 & 2 & \cdots & n \\
x_1 & x_2 & \cdots & x_n
\end{bmatrix} \implies \ell_{-1} H_i P_i \implies
\begin{bmatrix}
1 & 2 & \cdots & n \\
y_1 & y_2 & \cdots & y_n
\end{bmatrix}
$$

if and only if

$$(\text{Lang}(x_1), \text{Lang}(x_2), \ldots, \text{Lang}(x_n)) \implies \text{rew} \ (\text{Lang}(y_1), \text{Lang}(y_2), \ldots, \text{Lang}(y_n))$$

The initial configuration $S_0$ is chosen to be $S_0 = \begin{bmatrix}
1 & 2 & \cdots & n \\
L_1 & L_2 & \cdots & L_n
\end{bmatrix}$ and $\text{Lang}(L_i) = S_i$ where $s_0 = (S_1, \ldots, S_n)$ is the initial configuration of the CCNLP system. From this we get that sequences of computation steps performed by CCNLP systems $\Gamma$ and sequences of computation steps performed by the corresponding ANRS $AN$ have the following property:

$$(s_1, s_2, \ldots, s_n) = x_0 \implies \text{rew} \ x_1 \implies \text{comm} \ x_2 \cdots x_{k-1} \implies \text{rew} \ x_k$$

if and only if

$$S_0 = X_0 \implies \text{rew} \ X_1 \implies \text{comm} \ X_2 \cdots X_{k-1} \implies \text{rew} \ X_k$$

and for each $j = 0 \ldots k \ X_j = \begin{bmatrix}
1 & 2 & \cdots & n \\
x_1^j & x_2^j & \cdots & x_n^j
\end{bmatrix}$ and $x_j = (x_1^j, x_2^j, \ldots, x_n^j)$ and

$$\text{Lang}(x_i^j) = x_i^j$$

for all $i = 1 \ldots n$.

From the representation of general CCNLP systems it follows that a test-tubes system $\Gamma = (V, (V_1, R_1, A_1), \ldots, (V_n, R_n, A_n))$ can viewed as an ANRS $AN = (TG, Ag, D, S_0, Ag_{act}, P_{\text{comm}}, \{P_a | a \in Ag_{act}\})$ where $TG, Ag, D, Ag_{act}$ are the same as described above, $P_{\text{comm}}$ as described for the communication protocol $(a)$, $P_i = SP(V, R_i)$ as described in Subsection 5.3.2. In this case $P_i^{o, o' \leftarrow \overline{o}, o''}$ has the form

$$\{(\overline{0} \leftarrow [\overline{0}] \rightarrow [\overline{0} \leftarrow \overline{o}]\}$$

and $P_i^{\overline{0}, \overline{0' \leftarrow o'}, \overline{0'' \leftarrow o''}}$ might be implemented by

$$CHECK\{(\overline{0} \leftarrow [\overline{0}] \rightarrow [\overline{0} \leftarrow \overline{0}])\}$$

Now the representation of CCPC grammar systems as ANRS systems will be given. Here we will treat only CCPC systems with communication without splitting. There are at least two ways to represent CCPC grammar systems as ANRS systems. One of them is to represent string of each agent as a set of strings consisting of a single element. That is, the data of each agent is a singleton set of strings. This approach is very close to the representation of general CCNLP systems. Another way to rewrite CCPC grammars to ANRS systems
is to take labeled graphs representing strings for data objects and introduce passive agents as temporary storage places for strings being exchanged during the communication steps. So, formally the first representation looks like this.

**Definition 5.3.2.** Let $\Gamma = (N, T, (S_i, P_i, R_i), \ldots, (S_n, P_n, R_n))$ be a CCPC grammar system. Define the ANRS system corresponding to system $\Gamma$ as follows:

$$AN = (TG, Ag, D, S_0, Ag_{act}, P_{comm}, \{P_a | a \in V_{Ag_{act}}\})$$

where $TG = (\{1, 2, 3, \ldots, n\}, \emptyset, \emptyset, \emptyset)$ and $Ag = TG = Ag_{act}$ and $D = N_{LGraph.LG.Lab}$ where $Lab = \{i, \tilde{i}, i | i = 1 \ldots n\}$ and $LG = (Lab, Lab \times Lab, t, s)$, $t(i, j) = j$,

$s(i, j) = i$ for each $(i, j) \in Lab \times Lab$ and $S_0 = \begin{bmatrix} 1 & 2 & \cdots & n \\ S_1 & S_2 & \cdots & S_n \end{bmatrix}$.

For each agent $i = 1 \ldots n$ define $P_i$ as follows: $P_i = H_i AP_{P_i} \downarrow$. The program $P_{comm}$ which encodes the communication steps is a bit more complicated.

$$P_{ij} = H_i P_{R_j}^{i, i \rightarrow i},$$

$$CHECK(\begin{bmatrix} i \\ \tilde{i} \end{bmatrix}, COPY(i, i \rightarrow \tilde{i}, (j, \tilde{i}), SKIP)$$

$$NEW_i = H_i CHECK(\begin{bmatrix} j \end{bmatrix} | a \in N \cup T), CONCAT_i, NOINPUT_i)$$

$$CONCAT_i = DEL_i DEL_i; APP'_{1, i}; \ldots; APP'_{i-1, i}; APP'_{i+1, i}; APP'_{n, i}$$

$$APP'_{z, x} = CHECK(\{z\}, APP^{COPY}_{z, x}; DEL_{z, x}, SKIP)$$

$$NOINPUT_i = CHECK(\begin{bmatrix} i \leftarrow \tilde{i} \\ \emptyset \end{bmatrix}), DEL_{\tilde{i}}, FIN_i, SKIP)$$

$$DEL_{\tilde{i}} = \{(\begin{bmatrix} i \leftarrow \tilde{i} \\ \emptyset \end{bmatrix} \leftarrow (\begin{bmatrix} i \\ \emptyset \end{bmatrix}) \} \downarrow$$

$$FIN_i = \begin{cases} SKIP & \text{if } \Gamma \text{ is non-returning} \\ DEL_i; CREATE_{i, S_i} & \text{if } \Gamma \text{ is returning} \end{cases}$$

$$CREATE_{i, S_i} = \{(\begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix} \leftarrow (\begin{bmatrix} i \\ S_i \end{bmatrix}) \}$$

$$P_{comm} = \bigcup_{i, j=1, i \neq j}^{n} P_{ij} \downarrow ||_{i=1}^{n} NEW_i$$
The states of the ANRS system defined in Definition 5.3.2 are of the form

\[
\begin{bmatrix}
1 & 2 & \ldots & n \\
1 & 2 & \ldots & n \\
\bar{x}_1 & \bar{x}_2 & \ldots & \bar{x}_n
\end{bmatrix}
\]

(5.6)

which corresponds to the state \((x_1, x_2, \ldots, x_n)\) of the system \(\Gamma\).

The communication program \(P_{\text{comm}}\) works as follows. For each pair of agents \((i, j), i \neq j\) the program \(P_{ij}\) marks the string of the \(i\)th agent with \(i\) if it passes the input filter \(R_j\) of the \(j\)th agent. In this case it copies the string to the repository of the \(j\)th agent labeling it with label \(i\). Program \(NEW_i\) checks whether any non-empty string has been sent to the \(i\)th agent, that is, whether any subobject of the form \(\begin{bmatrix} j \end{bmatrix} \quad (a \in N \cup T)\) is present in the repository of the \(i\)th agent. If so, program \(CONCAT_i\) is executed, otherwise \(NOINPUT_i\) is executed. Program \(CONCAT_i\) concatenates strings with labels \(j = 1 \ldots n, j \neq i\) and replaces the original string of the \(i\)th agent \((\text{string labeled by } i)\) by the concatenation of strings with labels \(i = 1 \ldots n, j \neq i\). \(NOINPUT_i\) checks whether the original string of the agent has passed any of input filters of other agents (i.e. it checks whether there is any subgraph of the form \(i \leftarrow j\) present). If so, the original string of the \(i\)th agent is left unchanged if the system is non-returning, or replaced by \(S_i\) if the system is returning. If the string of the \(i\)th agent hasn’t passed the input filter of any of the agents then \(NOINPUT_i\) leaves the string unchanged. It is easy to see that

\[
\begin{bmatrix}
1 & 2 & \ldots & n \\
1 & 2 & \ldots & n \\
\bar{x}_1 & \bar{x}_2 & \ldots & \bar{x}_n
\end{bmatrix} \Rightarrow \begin{bmatrix} \bar{y}_1 & \bar{y}_2 & \ldots & \bar{y}_n \end{bmatrix}
\]

if and only if

\[(x_1, x_2, \ldots, x_n) \Rightarrow (y_1, y_2, \ldots, y_n)\]

By Subsection 5.3.2 we have that for each \(P_i, i = 1 \ldots n:\)

\[x_i \Rightarrow y_i \text{ if and only if } x_i \Rightarrow z \text{ and } \forall z : y_i \Rightarrow z\]

This implies that

\[
\begin{bmatrix}
1 & 2 & \ldots & n \\
1 & 2 & \ldots & n \\
\bar{x}_1 & \bar{x}_2 & \ldots & \bar{x}_n
\end{bmatrix} \Rightarrow \begin{bmatrix} \bar{y}_1 & \bar{y}_2 & \ldots & \bar{y}_n \end{bmatrix}
\]

if and only if

\[(x_1, x_2, \ldots, x_n) \Rightarrow (y_1, y_2, \ldots, y_n)\]

That is, the ANRS system \(AN\) defined in Definition 5.3.2 indeed simulates the work of the CCPC grammar system \(\Gamma\) and (5.6) gives the correspondence between the states of \(\Gamma\) and \(AN\).

Another approach to modeling CCPC grammar systems is the following
**Definition 5.3.3.** Let $\Gamma = (N,T,(S_i,P_i,R_i),\ldots,(S_m,P_m,R_m))$ be a CCPC grammar system. Define the ANRS system corresponding to system $\Gamma$ as follows:

$$AN' = (TG, Ag, D, S_0, Ag_{act}, P_{comm}, \{P_a | a \in V_{Ag_{act}}\})$$

where $TG = \{(i,i') | i = 1 \ldots n \} \cup \{(i,j) | i,j = 1 \ldots n, \emptyset, \emptyset \}$, $Ag = TG$, $Ag_{act} = \{(1,2,3,\ldots,n),\emptyset,\emptyset,\emptyset\}$ and $D = LGraph$.

The correspondence between states of $\Gamma$ and $AN'$ is the following: each state $(x_1, \ldots, x_n)$ of $\Gamma$ corresponds to the following state of $AN'$:

$$\begin{bmatrix} 1 & 2 & \cdots & n \\ x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

Programs performing rewriting can be defined as $P_i = AP_{P_i} \downarrow (i = 1 \ldots n)$.

The program $P_{comm}$ which simulates communication steps can be defined in the following way:

$$P_{ij} = P_{R_{ij}}^i(i,j),i$$

$$GATH_i = APP'_{(1,i),i'; \ldots; APP'_i(i-1,\emptyset),i'; \ldots; APP'_i(n,\emptyset),i';}$$

$$CHECK(\left\{ \begin{bmatrix} i' \\ a \end{bmatrix} | a \in N \cup T \right\}, \{DELi; \MOVE^i'; i, NOINP_i\})$$

$$NOINP_i = CHECK(\left\{ \begin{bmatrix} i, j \end{bmatrix} | a \in T \cup N \wedge 1 \leq j \leq n \right\}, \{FIN_i; SKIP\})$$

$$APP'_{x, z} = \left\{ \begin{bmatrix} z \\ \emptyset \end{bmatrix} \right\}, BAPP_{x, z}^{COPY}, SKIP$$

$$BAPP_{x, z}^{COPY} = CHECK(\left\{ \begin{bmatrix} z \\ \emptyset \end{bmatrix} \right\}, \{FIN_i; \{\emptyset \rightarrow \emptyset \})\}; APP_{x, z}^{COPY}$$

$$FIN_i = \left\{ \begin{array}{ll} SKIP & \text{if } \Gamma \text{ is non-returning} \\ DELi; CREATE_{i,S_i} & \text{if } \Gamma \text{ is returning} \end{array} \right\}$$

$$P_{comm} = \big\|_{i,j=1,i\neq j} P_{ij}^n;$$

$$\big\|_{i=1} GATH_i;$$

$$\big\|_{z \in \{(i,j),i', \mid i,j=1, \ldots, n\}} \text{DELi}$$

The program $P_{comm}$ works in the following way: $P_{ij}$ copies the string of the $i$th agent to the agent $(i,j)$ if the string passes the input filter of the $j$th agent. $GATH_i$ stores the concatenation of the message strings received by the
ith agent (strings of the agents \((j, i), j \neq i, j = 1 \ldots n\)) in the string of the agent \(i'\). If the agent \(i'\) contains the empty string or agent \(i'\) is not present at all, then it means that the agent \(i\) hasn’t received any messages. \(GATH_i\) checks whether this is the case, and if so, executes \(NOINP_i\), otherwise replaces the string of the agent \(i\) with the string of the agent \(i'\). \(NOINP_i\) checks whether the string of the \(i\)th agent has passed the input filters of any other agent (that is, there exists agent \((i, j)\) with a non-empty string). If so, then the string of the agent \(i\) either remains unchanged (if the system is non-returning) or replaced by \(S_i\) (if the system is returning). Otherwise the string of the agent \(i\) remains unchanged. Finally, all agents with label \((i, j)\) and \(i'\) are deleted together with their strings.

Again it is easy to see that

\[
\begin{bmatrix}
1 & 2 & \ldots & n
\end{bmatrix}
\xrightarrow{P_{comm}}
\begin{bmatrix}
y_1 & y_2 & \ldots & y_n
\end{bmatrix}
\]

if and only if

\[(x_1, x_2, \ldots, x_n) \Rightarrow_{comm} (y_1, y_2, \ldots, y_n)\]

and

\[
\begin{bmatrix}
1 & 2 & \ldots & n
\end{bmatrix}
\xrightarrow{\|_{H_i} P_i \|}
\begin{bmatrix}
y_1 & y_2 & \ldots & y_n
\end{bmatrix}
\]

if and only if

\[(x_1, x_2, \ldots, x_n) \Rightarrow_{rew} (y_1, y_2, \ldots, y_n)\]

That is, \(AN'\) indeed simulates the behavior of the CCPC grammar system \(\Gamma\).

### 5.3.4 PCLP systems

In this subsection the representation of PCLP systems by ANRS systems will be introduced. As a particular case, representation of PC grammar systems communicating by command will be given. Let be \(\Gamma = (V, K, \Pi_1, \ldots, \Pi_n)\) a PCLP system. In order to construct a ANRS system equivalent to the system \(\Gamma\) we will view the work of \(\Gamma\) as alternating sequences of rewriting and communication steps. As it was already mentioned in Chapter 3, by extending the definition of communication steps to the case when no queries are present (in this case the communication step is supposed to leave the state of the system unchanged) it is indeed possible to represent the work of a PCLP system by alternating sequences of rewriting and communication steps.

**Definition 5.3.4.** Let \(\Gamma = (V, K, \Pi_1, \ldots, \Pi_n)\) be a PCLP system. Define the **ANRS system corresponding to system \(\Gamma\)** in the following way

\[
AN = (TG, Ag, D, S_0, Agact, P_{comm}, \{P_a | a \in V_{Agact}\})
\]
5.3. NLP SYSTEMS AS ANRS SYSTEMS

where \( TG = \{1, \ldots, n, #, P\}, \{i \xrightarrow{C} j\} | i,j = 1 \ldots n, P\} \cup \{i \xrightarrow{\gamma} j\} | j = 1 \ldots n, x \in \{M, \mu\}\}, t_{TG}, \sigma_{TG}, t_{TG}(i \xrightarrow{\gamma} j) = j, \sigma_{TG}(i \xrightarrow{\gamma}) = i \) for each \( x \in \{M, C, \mu\}, Ag = \{1, \ldots, n\} \), \( Ag_{act} = Ag, D = \text{LGraph} \) and \( S_0 = \begin{bmatrix} 1 & 2 & \ldots & n \end{bmatrix} \).

In the construction presented below each state \((x_1, \ldots, x_n)\) of the PCLP system \( \Gamma \) corresponds to the state \(\begin{bmatrix} x_1 & x_2 & \ldots & x_n \end{bmatrix}\) We assume that for each \( i = 1 \ldots n \) program \( P_i \) simulates the work of the language processor \( \Pi_i = (V, M, \mu, \Pi) \) working in mode \( f_i \). That is, for each string \( x \in V^* \)

\[
y \in \Pi_i^f(\{x\}) \iff \overline{x} \implies P_i \overline{y}
\]

The concrete shape of the communication program \( P_{comm} \) depends on the communication protocol used by the particular PCLP system \( \Gamma \). Concrete versions of \( P_{comm} \) will be presented for each of communication protocols (a), (b), (c) below. In the following text we will use notation of the form \(\begin{bmatrix} K_1 \xrightarrow{\gamma} K_2 \end{bmatrix} \) where \( x \in \{M, \mu, C\} \) and \( K_1, K_2 \) are graphs containing only one agent labeled node each and the edge \( \gamma \) goes from a node of \( K_1 \) to a node of \( K_2 \). Then the notation above denotes object \((K, m, f)\) where \( K \) is the graph \( K_1 \) and \( K_2 \) together with the edge \( \gamma \), \( m \) the appropriate morphism to the typing graph, \( f(a_1) = x_1 \) and \( f(a_2) = x_2 \) where \( a_1 \) is the agent labeled node of \( K_1 \) and \( a_2 \) is the agent labeled node of \( K_2 \).

**protocol (a)**

\[
\text{BUILD}_i = \bigcap_{j=1}^{n} \{[i \xrightarrow{\gamma} j] \leftarrow [i \xrightarrow{C} j] \rightarrow [i \xrightarrow{C} j] \}
\]

\[
\text{REPLY}_i = \text{CHECK}(\{[i \xrightarrow{C} j] | j = 1 \ldots n\}, \text{SKIP}, \text{SEND}_i)
\]

\[
\text{SEND}_i = \bigcap_{j=1}^{n} \{[i \xrightarrow{\gamma} j] \leftarrow [i \xrightarrow{\gamma} j] \rightarrow [i \xrightarrow{\gamma} j] \}
\]

\[
\text{COPY}^{i,j} = \{[i \xrightarrow{\gamma} j] \leftarrow [i \xrightarrow{\gamma} j] \rightarrow [i \xrightarrow{\gamma} j] \}; \text{COPY}^{j,i} \}
\]

\[
\text{CONSUME}_i = \text{CHECK}(\{[i \xrightarrow{\gamma} j] | j = 1 \ldots n\}, \text{SKIP}, \bigcap_{j=1}^{n} \text{COPY}^{j,i})
\]

\[
\text{RET}_i = \text{CHECK}(\{[i \xrightarrow{\gamma} j] | j = 1 \ldots n\}, \text{FIN}_i, \text{SKIP})
\]
\[ \text{FIN}_i = \bigcap_{j=1}^n \{ ([i \downarrow j] \leftarrow [i \downarrow j] \rightarrow [i \downarrow j]) \}; \]

\[ \text{FIN}'_i = \begin{cases} 
H_i\text{DELE} ; \{ ([i] \leftarrow [i]) \rightarrow [i]) \} & \text{if } \Gamma \text{ is returning} \\
\text{SKIP} & \text{if } \Gamma \text{ is non-returning} 
\end{cases} \]

\[ \text{P}_{\text{com}} = \text{CHECK}(\{ [i] \mid 1 \leq i, j \leq n \}; \bigcap_{i=1}^n \text{BUILD}_i; \text{COMM} \downarrow, \text{SKIP}) \]

\[ \text{PCHEC} = \{ ([i] \leftarrow [i] \rightarrow [i]) \mid 1 \leq i, j \leq n \} \]

\[ \text{COMM} = \text{PCHEC}; \bigcap_{i=1}^n \text{REPLY}_i; \bigcap_{i=1}^n \text{CONSUME}_i; \bigcap_{i=1}^n \text{RET}_i \]

\[ \text{DEL} = \{ \text{del}_{a,b} \mid a, b \in V \} \downarrow \]

\[ \text{del}_{a,b} = ((a \rightarrow b) \leftarrow (a \rightarrow b)) \]

\[ \text{del}_a = (a \leftarrow \emptyset \rightarrow \emptyset) \]

\( \text{P}_{\text{com}} \) works in the following way. If no query is present in the system then the program doesn’t change the state object which it has been applied to. Otherwise it executes programs \( \text{BUILD}_i \) in parallel manner and afterwards it executes program \( \text{COMM} \) as many times as possible.

For each agent \( i \) program \( \text{BUILD}_i \) adds an edge labeled by \( C \) pointing from the node labeled by \( i \) to the node labeled by \( j \) if and only if the string of the agent \( i \) contains the query symbol \( Q_j \). Program \( \text{COMM} \) does the rest of communication. In fact, program \( \text{COMM} \) corresponds to consecutively executed communication steps of the system \( \Gamma \). It starts with executing program \( \text{PCHEC} \). Program \( \text{PCHEC} \) can be applied to a state object if and only if at least one of the agents’ string contains query symbols. If it can be applied, it doesn’t change the object which it has been applied to. For each agent \( i \) program \( \text{REPLY}_i \) notifies all those agents which are expecting information from the \( i \)th agent that the agent \( i \) is ready to give this information, that is, the string of the agent \( i \) doesn’t contain any query symbols. This is equivalent to the condition that there is no edge pointing from the node labeled by \( i \) which has been introduced by the program \( \text{BUILD}_i \) (i.e no edge labeled by \( C \)) . The notification is done by replacing each edge labeled by \( C \) pointing from a node labeled by some \( j \) to the node labeled by \( i \) by and edge pointing from \( i \) to \( j \) with label \( M \). Program \( \text{CONSUME}_i \) does the following: it checks whether all agents from which the agent \( i \) expects information are ready to give it. That is, whether there is no edge with label \( C \) pointing from \( i \) to some
5.3. NLP SYSTEMS AS ANRS SYSTEMS

If this is the case, then \( \text{COPY}^{j,i} \) is executed in parallel manner for all agents \( j \). Program \( \text{COPY}^{j,i} \) replaces the symbol \( Q_j \) in the string of the agent \( i \) if there is an edge labeled by \( M \) from the node labeled by \( j \) to the node labeled by \( i \). Besides \( \text{COPY}^{j,i} \) changes the label of this edge to \( \mu \). Edges with label \( \mu \) are intended to indicate that the agent corresponding to the source of the edge has sent its string to some other agents. Program \( \text{RET}_i \) changes the string of the \( i \)th agent if the \( i \)th agent has sent its string to some other agents. For each agent \( i \) program \( \text{RET}_i \) checks whether the agent \( i \) has sent anything to some other agents. If so, then depending on whether the system is returning or non-returning the string of the agent \( i \) is reset to the initial axiom \( S_i \) or kept unchanged, besides all edges pointing from \( i \) which are labeled by \( \mu \) are deleted. Program \( \text{COMM} \) is executed until all queries have been satisfied or forever, if the configuration is a blocking one. Program \( \text{DEL} \) deletes the string representation of an arbitrary string over \( V \).

**Protocol (b)**

\[
\text{CONSUME}_i = \prod_{j=1}^n \text{COPY}^{j,i}
\]

\[
P_{\text{comm}} = \text{CHECK}([i_{Q_j}| 1 \leq i, j \leq n] \prod [\text{BUILD}_i; \text{COMM} \downarrow; \text{SKIP}]
\]

\[
\text{COMM} = P_{\text{CHECK}}; \prod_{i=1}^n \text{REPLY}_i; \prod_{i=1}^n \text{CONSUME}_i; \prod_{i=1}^n \text{RET}_i
\]

Here programs \( \text{BUILD}_i, \text{REPLY}_i, P_{\text{CHECK}} \) and \( \text{RET}_i \) are the same as in the case of the protocol (a). Unlike for the protocol (a), the program \( \text{CONSUME}_i \) doesn’t check whether all queries of the agent \( i \) can be satisfied, but simply gets information from all those agents, which are ready to give it. All other aspects of communication are the same as for the protocol (a).

**Protocol (c)**

\[
\text{CHOOSE} = \{(\emptyset \leftarrow \emptyset \rightarrow \#)\}; \text{CHD} \downarrow;
\]

\[
\prod_{i,j=1}^n \{(P \subseteq i \emptyset \jmath \emptyset) \leftarrow (P \subseteq i \emptyset \jmath \emptyset) \rightarrow (P \subseteq i \emptyset \jmath \emptyset)
\}
\]

\[
\prod_{i=1}^n \{(P \subseteq i \emptyset) \leftarrow (i \emptyset) \rightarrow (i \emptyset)\}
\]
\[ \text{CHD} = \{( \# \ i \ \{ \ j \} \rightarrow [ \begin{array}{c} \# \\ i \\ 0 \\ 0 \end{array} ; \begin{array}{c} \# \\ i \\ 0 \\ 0 \end{array} ] \} | i, j = 1 \ldots n \}; \]

\[ \{( \begin{array}{c} P \{ \ j \} \rightarrow [ \begin{array}{c} \# \\ i \\ 0 \\ 0 \end{array} ; \begin{array}{c} \# \\ i \\ 0 \\ 0 \end{array} ] \} | i = 1 \ldots n \} \} \}

\[ P_{\text{comm}} = \langle \widehat{\text{BUILD}}_{i=1}^n ; \text{COMM} \downarrow \rangle \]

\[ \text{COMM} = \langle \text{CHOOSE} ; \widehat{\text{CONSUME}}_{i=1}^n ; \widehat{\text{RET}}_{i=1}^n \rangle \]

\[ \text{CONSUME}_i = \langle \widehat{\text{COPY}}_{j=1}^i \rangle \]

Here programs \( \text{RET}_i, \text{CONSUME}_i \) and \( \text{BUILD}_i \) are the same as for the protocol (b). Program \( \text{CHOOSE} \) chooses a query symbol \( Q_i \) satisfying the conditions specified by the protocol (c). That is, there should be some queries directed to the agent \( i \) (which is equivalent to presence of an edge from some \( j \) to \( i \)), and the string of the agent \( i \) shouldn’t contain query symbol \( Q_i \). Agent \( i \) whose query symbol has been chosen is marked by adding a node labeled by label \( P \) and an edge from that node to the node labeled by \( i \). Besides, \( \text{CHOOSE} \) replaces edges pointing to the chosen agent by edges labeled by \( M \) and pointing from the chosen agent to indicate that the chosen agent is ready to send its string wherever it is required. Since the string sent by the agent might contain further query symbols, programs \( \text{BUILD}_i \) must be executed again. All other parts of the program \( P_{\text{comm}} \) are the same as for the protocol (b).

It is easy to see from the description above that for each protocol (a), (b) or (c)

\[
\begin{bmatrix}
1 \\
x_1 \\
x_2 \\
\vdots \\
x_n
\end{bmatrix} \Rightarrow P_{\text{comm}} \begin{bmatrix}
1 \\
y_1 \\
y_2 \\
\vdots \\
y_n
\end{bmatrix}
\]

if and only if

\[
(x_1, x_2, \ldots, x_n) \Rightarrow^*_{\text{comm}} (y_1, y_2, \ldots, y_n) \text{ and } |y_i|_K = 0 \text{ for all } 1 \leq i \leq n
\]

provided that there exists such a \( i = 1 \ldots n \) that \( |x_i|_K \neq 0 \). Recall, that

\[
(x_1, x_2, \ldots, x_n) \Rightarrow_{\text{comm}} (y_1, y_2, \ldots, y_n) \text{ and } \forall i = 1 \ldots n : |x_i|_K = 0 \Rightarrow y_i = x_i(i = 1 \ldots n)
\]

But in this case

\[
\begin{bmatrix}
1 \\
x_1 \\
x_2 \\
\vdots \\
x_n
\end{bmatrix} \Rightarrow P_{\text{comm}} \begin{bmatrix}
1 \\
y_1 \\
y_2 \\
\vdots \\
y_n
\end{bmatrix}
\]

implies that \( x_i = y_i \) for all \( i = 1 \ldots n \). That is, a communication step of the ANRS system \( \Gamma \) indeed corresponds to a maximal sequence of consecutive communication steps of the PCLP system \( \Gamma \).
5.3. NLP SYSTEMS AS ANRS SYSTEMS

Rewrite steps of the original PCLP system $\Gamma$ and the equivalent ANRS system $\text{AN}$ correspond to each other in the following sense.

$$\begin{bmatrix} 1 & 2 & \cdots & n \\ x_1 & x_2 & \cdots & x_n \end{bmatrix} \implies \|_{-1}^n H_i P_i \begin{bmatrix} 1 & 2 & \cdots & n \\ y_1 & y_2 & \cdots & y_n \end{bmatrix}$$

if and only if

$$(x_1, x_2, \ldots, x_n) \implies_{\text{rew}} (y_1, y_2, \ldots, y_n) \text{ and } |x_i|_K = 0$$

That is, the ANRS system $\text{AN}$ indeed simulates the work of the PCLP system $\Gamma$ and a state $\begin{bmatrix} 1 & 2 & \cdots & n \\ x_1 & x_2 & \cdots & x_n \end{bmatrix}$ of $\text{AN}$ corresponds to the state $(x_1, x_2, \ldots, x_n)$ of the system $\Gamma$. That is,

$$x_0 = (S_1, S_2, \ldots, S_n) \implies_{\text{rew}} x_1 \implies_{\text{comm}} \cdots \implies_{\text{rew}} x_k$$

if and only if

$$X_0 = S_0 \implies_{\text{rew}} X_1 \implies_{\text{comm}} \cdots \implies_{\text{rew}} X_k$$

where $X_j = \begin{bmatrix} 1 & 2 & \cdots & n \\ x_1^j & x_2^j & \cdots & x_n^j \end{bmatrix}$ and $x_j = (x_1^j, x_2^j, \ldots, x_n^j) (j = 1 \ldots n)$.

Particularly, if we take a PC grammar system $\Gamma = (N, T, K, G_1, \ldots, G_n)$ and we construct the corresponding ANRS system $\text{AN}$ as defined in Definition 5.3.4 and we define $P_{\text{comm}}$ to be of the shape described above and we take $P_i := AP_{P_i}$ for each component $i = 1 \ldots n$ with $G_i = (N, T, S_i, P_i)$ then the resulting ANRS system $\text{AN}$ will simulate the work of the PC grammar system $\Gamma$. 
CHAPTER 5. ABSTRACT NETWORKS OF REWRITING SYSTEMS
Chapter 6

Conclusions and future work

In this work a generalization of the notion of network of language processors has been introduced. Abstract networks of rewriting systems defined in this work provide not only a generalization of networks of language processors, they also provide a coherent formalisms for describing such systems and their generalizations. As the material presented in Section 5.3 shows, abstract networks of rewriting systems give a framework suitable for describing several models of networks of language processors in a uniform way. As a side effect of attempts to find a universal formalization of NLP systems two constructions, important for their own sake, have been developed in this work. One of them is the concept of high-level replacement programs. As far as the author knows, it is the first attempt to define programmed rewriting for high-level replacement, that is, programmed rewriting of objects of an arbitrary category. The concept presented in this work is a generalization of concepts from [14, 25]. Besides, high-level replacement programs can be executed in parallel, so their expressive power exceeds that of presented in [14]. Another important "side-effect" of this thesis is the concept of ANRS state categories. This construction can be viewed as a generalization of the concept of hierarchical graphs. For example, it is possible to construct such a ANRS category that resembles hierarchical hypergraphs of a fixed depth defined in [21].

The power of abstract networks of rewriting systems lies in the extensive use of category theory. Due to the expressive power of category theory it is possible to describe quite different systems by using the framework of abstract networks of rewriting systems. Further research might be directed towards finding a more general description of states of a multi-agent system. Namely, instead of using graphs for encoding communication of agents it could be reasonable to use objects of an arbitrary category. It would be also interesting to extend the approach described in this work to those models of multi-agent systems which encode interaction of agents by manipulations of objects shared
by all agents. Nice examples of such systems are eco-grammars, PM-colonies and CD grammars. It might be also interesting to describe actor systems and membrane computers within the framework of ANRS systems. Although these two systems were inspired by completely different phenomena, they share some common features as well. Investigating the possibility of using ANRS systems for describing models of molecular computers might be an intriguing topic.

Another important contribution of this work is the definition of programmed rewriting of objects of an arbitrary category. It turned out that it is possible to define the semantics of HLRP programs via derivation sequences in such a way that HLRP conditions 5, 6, 7 are not needed for proving the correctness of those definitions. Besides, for HLRP categories that definition of semantics of HLRPs is equivalent to the definition of semantics of HLRPs that was given in this work.

High-level replacement programs rise several questions worth of studying. One of them is the relationship between high-level replacement programs and high-level replacement systems. Some results about simulating HLRPs by high-level replacement systems might turn out to be useful.

As the few ideas mentioned above reflect, there are quite many interesting questions which are waiting for further investigation.
Appendix A

Properties of ANRS state categories

The aim of this appendix is to give the proofs of the theorems stated in Section 5.2.1. The first section gives the proof of Theorem 5.2.1. The second section gives the limit and colimit constructions in the category of ANRS states. The last section contains the proof of Theorem 5.2.2

A.1 ANRS states as categories

Proof of Theorem 5.2.1. It is easy to see that arrows and objects in the construction of $N_{D,TG,A_2}$ are well defined. What remains to be proven is that axioms of a category hold for $N_{D,TG,A_2}$:

if $\text{codom}(h) = \text{dom}(g)$ then $g \circ h$ is defined

Indeed, let $h : O_1 \to O_2$ and $g : O_2 \to O_3$ be arrows where $O_i = (G_i, m_i : G_i \to TG, f_i : V_{m_i^{-1}(A_2)} \to \text{Obj}(D))$. Then assuming that $h = (h_G, \{h_v | v \in V_{m_1^{-1}(A_2)}\})$ and $g = (g_G, \{g_v | v \in V_{m_2^{-1}(A_2)}\})$ we get that $h_G : G_1 \to G_2$ and $g_G : G_2 \to G_3$ so $g_G \circ h_G$ is well defined and $m_3 \circ g_G \circ h_G = m_1$. Besides $h_v : f_1(v) \to f_2(h_G(v))$ and $g_w : f_2(w) \to f_3(g_G(w))$ for all $v \in V_{m_1^{-1}(A_2)}$ and $w \in V_{m_2^{-1}(A_2)}$. That is, $g_{h_0(v)} \circ h_v : f_1(v) \to f_3(g_G \circ h_G(v))$ is well defined for each $v \in V_{m_1^{-1}(A_2)}$.

$\text{codom}(g \circ h) = \text{codom}(g)$ and $\text{dom}(g \circ h) = \text{dom}(g)$  Immediate from the proof of the previous item.

Composition of arrows is associative  Follows from the associativity of composition of graphs morphisms and arrows of $D$.

Existence of identity arrows and their properties  Immediate from the definition of identity arrows and composition

\qed

99
A.2 Construction of limits and colimits

Let $\Delta$ be a finite diagram over $N_{D,TG,Ag}$. We will view finite diagrams over a category as graphs whose nodes are objects of the category and whose edges are arrows of the category. The source node of an edge is the domain of the corresponding arrow of the category, the target node is the codomain of the corresponding arrow of the category. Denote by $\Delta_i$ the $i$th node of the diagram $\Delta$ ($1 \leq i \leq n$). Denote by $\Delta_{i,j,k}$ the $k$th arrow from the node $\Delta_i$ to the node $\Delta_j$ ($1 \leq k \leq n_i$ and $n_i \geq 0$) in the diagram $\Delta$. Assume that for each $i,j,k$ $\Delta_i = (G_i, m_i, g_i)$ and $\Delta_{i,j,k} = (h_{i,j,k}, \{h_{i,j,k}, v \in V_{m_i^{-1}(Ag)}\})$. Then let $\Delta_G$ be the following diagram over the category of graphs typed over $TG$: $\Delta_G = \{\Delta_i[i = 1 \ldots n], \{h_{i,j,k}[i, j = 1 \ldots n, k = 1 \ldots n_i]\}, t_G, s_G\}$ where $s_G(h_{i,j,k}) = i$ and $t_G(h_{i,j,k}) = j$. That is, the diagram $\Delta_G$ consists of the communication graph components of the nodes of the diagram $\Delta$ connected by the appropriate component of the morphisms of the diagram $\Delta$.

Theorem A.2.1. Let $\Delta$ be a finite commutative diagram in $N_{D,TG,Ag}$. Let $Z = (G_Z, m_Z, f_Z)$ be constructed in the following way

- (a) The typed graph $(G_Z, m_Z)$ together with the morphisms $f_i (1 \leq i \leq n)$ is the colimit of the diagram $\Delta_G$.
- (b) For arbitrary $v \in V_{m_i^{-1}(Ag)}$ let $K_v = \{i_1, \ldots, i_{k_v}\} \subseteq \{1, 2, \ldots, n\}$ be such a set that $v_{ij} = f_i^{-1}(v)$ and for all $i \notin K_v$ the inverse image $f_i^{-1}(v)$ is undefined. Then $f_Z(v)$ together with arrows $f_{v_{ij}}$ is the colimit of the diagram $\Delta_v = (K_v, \{h_i, j, k, v[i, j \in K_v, 1 \leq k \leq n_i]\}, t_v, s_v)$ where $t_v(h_i, j, k, v[i]) = v_{ij}$ and $s_v(h_i, j, k, v[i]) = v_{ij}$.

Then $Z$ together with morphisms $z_i = (f_i, \{f_i, v \in V_{m_i^{-1}(Ag)}\})$ is the colimit of the diagram $\Delta$.

Proof. First of all, since $G_Z$ is the colimit of $\Delta_G$ we have that for each $i = 1 \ldots n$ and $v \in V_{m_i^{-1}(Ag)}$ the arrow $f_{v,i}$ is well defined. Besides,

$$z_j \circ \Delta_{i,j,k} = (f_j \circ h_{i,j,k}[f_i, h_{i,j,k}[v] \in V_{m_i^{-1}(Ag)}]) = (f_i, \{f_i, v \in V_{m_i^{-1}(Ag)}\})$$

because of the assumption that $(G_Z, m_Z)$ with $f_i$ is the colimit, therefore a cocone of $\Delta_G$ and $f_Z(f_i(v)) = f_Z(f_j(h_{i,j,k}(v)))$ together with morphisms $f_{v,i}$ is the colimit, therefore a cocone of $\Delta_{f_i(v)}$ for all $v \in V_{m_i^{-1}(Ag)}$. That is, $Z$ is a cocone of the diagram $\Delta$.

$Z$ together with morphism $z_i (i = 1 \ldots n)$ is the colimit of the diagram $\Delta$. Indeed, let $H = (G_H, m_H, f_H)$ be together with morphisms $\pi_i = (\mu_i, \{\mu_v, v \in V_{m_i^{-1}(Ag)}\}) : \Delta_i \to H$ a cocone of the diagram. Then $(G_H, m_H)$ with morphisms $\mu_i$ is the cocone of $\Delta_G$. That means that there is a unique morphism $u_G : (G_Z, m_Z) \to (G_H, m_H)$ such that $u_G \circ f_i = \mu_i$. Besides, for each $v \in V_{m_i^{-1}(Ag)}$ we have that $f_H(u_G(v))$ together with arrows $\mu_{v_{ij}}$ ($i,j \in K_v$) is a cocone of
the diagram $\Delta_i$. That means that there exists a unique arrow $u_v : f_Z(v) \to f_H(u_G(v))$ such that $u_v \circ f_{i,j,v_i} = \mu_{i,j,v_j}$ for each $i,j \in K_v$.

That is, there exists an arrow $u = (u_G, \{u_v | v \in V_{m^{-1}}(A_g)\}) : Z \to H$ such that $u \circ z_i = \pi_i$. It is easy to see that $u$ is unique. Indeed, assume that there exists an arrow $\tilde{u} = (\tilde{u}_G, \{\tilde{u}_v | v \in V_{m^{-1}}(A_g)\}) : Z \to H$ such that $\tilde{u} \circ z_i = \pi_i$. But then $\tilde{u}_G \circ f_i = \mu_i$ from which because of the uniqueness of $u_G$ we get that $\tilde{u}_G = u_G$. It is easy to see that $\tilde{u} \circ z_i = \pi_i$ implies that $\tilde{u}_v \circ f_{i,j,v_i} = \mu_{i,j,v_j}$ for each $v \in V_{m^{-1}}(A_g)$ which implies -- because of the uniqueness of $u_v$ -- that $\tilde{u}_v = u_v$ for each $v \in V_{m^{-1}}(A_g)$. That is, $\tilde{u} = u$.

We have just proven that $Z$ together with morphisms $z_i$ is indeed the colimit of the diagram $\Delta$.

A similar statement can be proven for limits.

**Theorem A.2.2.** Let $\Delta$ be a finite commutative diagram in $N_{D,TG,A_g}$. Let $Z = (G_Z, m_Z, f_Z)$ be constructed in the following way

- (a) The typed graph $(G_Z, m_Z)$ together with the morphisms $f_i (1 \leq i \leq n)$ is the limit of the diagram $\Delta_G$.

- (b) For arbitrary $v \in V_{m^{-1}}(A_g)$ let $K_v \subseteq \{1, \ldots, n\}$ be such a set that $v_j = f_j(v) \iff j \in K_v$. Then $f_Z(v)$ together with arrows $f_{v,j}$ is the limit of the diagram $\Delta_v = (K_v, \{h_{i,j,k,v_i} | i,j \in K_v, 1 \leq k \leq n_i\}, t_v, s_v)$ where $t_v(h_{i,j,k,v_i}) = v_j$ and $s_v(h_{i,j,k,v_i}) = v_i$.

Then $Z$ together with morphisms $z_i = (f_i, \{f_{i,v} | v \in V_{m^{-1}}(A_g)\})$ is the limit of the diagram $\Delta$.

**Proof.** First of all, arrows $f_{v,j}$ are well defined for each $v \in V_{m^{-1}}(A_g)$. Besides, $h_{i,j,k} \circ f_i = f_j$ and $h_{i,j,k,f_{i,v}} \circ f_{i,v} = f_{v,j}$ for all $v \in V_{m^{-1}}(A_g)$ and $1 \leq i, j \leq n$ and $1 \leq k \leq n_i$. That is, $z_i \circ \Delta_{i,j,k} = z_j$, which means that $Z$ with morphisms $z_i$ is indeed a cone of the diagram $\Delta$.

As the second step we will prove that $Z$ together with morphisms $z_i$ is indeed a limit. Consider an object $H = (G_H, m_H, f_H)$ together with arrows $\pi_i = (\mu_i, \{\mu_{i,v} | v \in V_{m^{-1}}(A_g)\})$ such that $H$ is a cone of the diagram $\Delta$. From this it follows that $(G_H, m_H)$ with arrows $\mu_i$ is a cone of the diagram $\Delta_G$. It implies that there exists a unique arrow $u_G : (G_H, m_H) \to (G_Z, m_Z)$ such that $f_i \circ u_G = \mu_i$. But for all $w \in V_{m^{-1}}(A_g)$ we have that $f_Z(u_G(w))$ with arrows $f_{u_G(v),i}$ is the limit of $\Delta_{u_G(v)}$ and $f_H(w)$ with arrows $\mu_{i,w}$ is a cone of the diagram $\Delta_{u_G(w)}$. This implies that there exists a unique arrow $u_w : f_H(w) \to f_Z(u_G(w))$ such that $f_{u_G(v),i} \circ u_w = \mu_{i,w}$. That is, there exists uniquely an arrow $u = (u_G, \{u_w | w \in V_{m^{-1}}(A_g)\})$ such that $z_i \circ u = \pi_i$ for all $i = 1 \ldots n$. That is, $Z$ together with morphisms $z_i$ is indeed the limit of diagram $\Delta$. 

\[ \square \]
A.3 Inheritance of HLRP conditions

In this section the proof of Theorem 5.2.2 will be given. To do this, the inheritance of each property will be proven separately. The proof goes as follows: we assume that the category $D$ with class of arrows $M_D$ satisfies HLRP conditions. Using this assumption it will be proven that each of the HLRP conditions holds for the category $N_{T_G,A_B,D}$ with class of arrows $M$ as defined in Theorem 5.2.2. We will go through HLRP conditions in the same order as they were introduced in Definition 4.2.1.

1. Assume that $A = (G_A,m_A,f_A), B = (G_B,m_B,f_B)$ and $C = (G_C,m_C,f_C)$ are arbitrary objects and $f : A \to B$ and $g : A \to C$ are arbitrary arrows and $f$ belong to $M$. By the definition of the class $M$ we get that for each $v \in V_{m_A^{-1}(A_B)}$ the arrow $f_v$ is in $M_D$. Consider the object $D = (G_D,m_D,f_D)$ and arrows $h : C \to D$ and $k : B \to D$ such that $G_D$ with arrows $k_G$ and $h_G$ is the pushout of $G_C \leftarrow G_A \rightarrow G_B$ and for each $v \in V_{m_A^{-1}(A_B)}$ the object $k_{f_D(v)}(f_G(v))$ together with morphisms $k_{f_D(v)}$ and $h_{f_D(v)}$ is the pushout of $f_B(f_G(v)) \xrightarrow{f_A(v)} f_C(g_G(v))$.

For all $v \in V_{m_A^{-1}(A_B)} \setminus f_G(V_{G_A}) (v \in V_{m_C^{-1}(A_B)} \setminus g_G(V_{G_A}))$ let $k_v$ (respectively $h_v$) be $id_{f_D(v)}$ (id$_{f_C(v)}$ respectively). According to the Theorem A.2.1 the object $D$ with $h$ and $k$ is indeed a pushout. Besides, since $f_v$ belongs to $M_D$ we get that $h_{f_D(v)}$ belongs to $M_D$ for each $v \in V_{m_A^{-1}(A_B)}$ and $id_a$ is in $M_D$ for each $w \in V_{m_C^{-1}(A_B)} \setminus g_G(V_{G_A})$. Thus we get that $h$ is in $M$.

2. Consider objects $B = (G_B,m_B,f_B)$, $C = (G_C,m_C,f_C)$ and $D = (G_D,m_D,f_D)$ and morphisms $h : B \to D$ and $k : C \to D$. Assume that $h$ and $k$ are in $M$. Consider the object $A = (G_A,m_A,f_A)$ and morphisms $f : A \to B$ and $g : A \to C$ such that $(G_A,m_A)$ with morphisms $f_G$ and $g_G$ is the pullback of $(G_B,m_B) \xrightarrow{h} (G_D,m_D) \xrightarrow{k} (G_C,m_C)$ and for each $v \in V_{m_A^{-1}(A_B)}$ the object $f(v)$ with morphisms $f_v$ and $g_v$ is the pullback of $f_B(f_G(v)) \xrightarrow{f_A(v)} f_C(g_G(v))$.

Since for all $v \in V_{m_A^{-1}(A_B)}$ arrows $h_{f_D(v)}$ and $k_{f_D(v)}$ are in $M_D$, the pullback of $f_B(f_G(v)) \xrightarrow{h_{f_D(v)}} f_D(h_G(f_G(v))) \xrightarrow{k_{f_D(v)}} f_C(g_G(v))$ exists and $f_v$ and $g_v$ are in $M$. Besides, according to the Theorem A.2.2 the object $A$ with morphisms $f$ and $g$ is indeed the pullback of $B \xrightarrow{h} D \xrightarrow{k} C$. Besides, since for all $v \in V_{m_A^{-1}(A_B)}$ morphisms $f_v$ and $g_v$ are in $M_D$, morphisms $f$ and $g$ are in $M$ too.
2. Consider the following pushout diagram:

\[
\begin{array}{c}
A \xrightarrow{f} B \\
\downarrow g & \downarrow h \\
C \xrightarrow{k} D
\end{array}
\]

Then the diagram

\[
\begin{array}{c}
G_A \xrightarrow{f_G} G_B \\
\downarrow g_G & \downarrow h_G \\
G_C \xrightarrow{k_G} G_D
\end{array}
\]

is a pushout and a pullback at the same time. For all \( v \in V_{m_A^{-1}(A_B)} \) we also have that

\[
\begin{array}{c}
f_A(v) \xrightarrow{f_v} f_B(f_G(v)) \\
\downarrow g_v & \downarrow h_{f_G(v)} \\
f_C(g_G(v)) \xrightarrow{k_G(g_G(v))} f_D(h_G(f_G(v)))
\end{array}
\]

is a pushout and since \( k_v, f_v, h_{f_G(v)}, k_G(g_G(v)) \) are in \( M_D \) the diagram above is also a pullback. Because of Theorem A.2.2 it means that the diagram (1) is also a pullback.

3. The binary coproduct of two objects \( A = (G_A, m_A, f_A) \) and \( B = (G_B, m_B, f_B) \) is the object \( A + B = (G_A + G_B, m_A + m_B, f_A + f_B) \) where \((G_A + G_B, m_A + m_B)\) is the coproduct of \((G_A, m_A)\) and \((G_B, m_B)\) and

\[
[f_A + f_B = \begin{cases} 
  f_A(v) & \text{if } v \in V_{m_A^{-1}(A_B)} \\
  f_B(v) & \text{if } v \in V_{m_B^{-1}(A_B)}
\end{cases}
\]

If \( f : A \to B \) and \( g : A' \to B' \) then the arrow \( f + g : A + A' \to B + B' \) is of the form \( f + g = \{ f_G + g_G \} \cup \{ f_v | v \in V_{m_A^{-1}(A_B)} \} \cup \{ g_w | w \in V_{m_B^{-1}(A_B)} \} \). If \( f \) and \( g \) are in \( M \), it implies that \( f_v \) and \( g_w \) are in \( M_D \) for all \( v \in V_{m_A^{-1}(A_B)} \) and \( w \in V_{m_B^{-1}(A_B)} \). That is, we get that \( f + g \) is in \( M \).
4. Consider the following diagram

\[
\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{g} & & \downarrow{\alpha} \\
C & \xrightarrow{l} & D \\
\downarrow{h} & & \downarrow{k} \\
E & \xrightarrow{j} & F
\end{array}
\]

where \( f, h, l, k \) and \( j \) are in \( M \) and (1) + (2) is a pushout and (2) is a pullback. Then in the following diagram the diagram (3) + (4) will be pushout and the diagram (4) will be a pullback. But since the category of typed graphs satisfies HLRP conditions we get that (3) and (4) are pushouts.

\[
\begin{array}{c}
(G_A, m_A) \xrightarrow{f_G} (G_B, m_B) \\
g_G \downarrow & & \downarrow \alpha_G \\
(G_C, m_C) & \xrightarrow{l_G} & (G_D, m_D) \\
h_G \downarrow & & \downarrow k_G \\
(G_E, m_E) & \xrightarrow{j_G} & (G_F, m_F)
\end{array}
\]

For all \( v \in V_{m_A^{-1}(A_B)} \) consider the following diagram

\[
\begin{array}{ccc}
f_A(v) & \xrightarrow{f_v} & f_B(f_G(v)) \\
g_v \downarrow & & \downarrow \alpha_{f_G(v)} \\
f_C(g_G(v)) & \xrightarrow{l_{g_G(v)}} & f_D(\alpha_{f_G(v)}) \\
h_{g_G(v)} \downarrow & & \downarrow k_{\alpha_{f_G(v)}} \\
f_E(h_G(g_G(v))) & \xrightarrow{j_{h_G(g_G(v))}} & f_F(k_{\alpha_{f_G(v)}}(f_G(v))))
\end{array}
\]
A.3. INHERITANCE OF HLRP CONDITIONS

Here (5) + (6) is pushout and (6) is a pullback. Then (5) and (6) are pushouts too. For all \( v \in V_{m \rightarrow 1}(A_g) \setminus g(g(V_{G_A})) \) by Theorem A.2.2 we have that the following diagram is a pullback:

\[
\begin{array}{ccc}
  f_C(v) & \xrightarrow{l_v} & f_D(l_G(v)) \\
  h_v \downarrow & & \downarrow k_{G(v)} \\
  f_E(h_G(v)) & \xrightarrow{j_{G(v)}} & f_F(j_G(h_G(v)))
\end{array}
\]

Since (1)+(2) is a pushout, by the colimit construction we get \( f_E(h_G(v)) \cong f_F(j_G(h_G(v))) \) which implies that \( l_v = \text{id}_{f_C(v)} \), \( f_C(v) = f_D(l_G(v)) \) and \( h_v = \phi^{-1} \circ k_{G(v)} \) where \( \phi : f_E(h_G(v)) \to f_F(j_G(h_G(v))) \) is the isomorphism are a good choice to make the diagram (7) to be a pullback.

On the other hand, for all \( v \in V_{m \rightarrow 1}(A_g) \setminus f_G(V_{G_A}) \) we get (by using Theorem A.2.1) that \( \mu^{-1} \circ k_{G(v)} \circ o_v = id_{f_C(v)} \) where \( \mu = k_{G(v)} \circ o_v \) is an isomorphism. Since \( k_{G(v)} \) is in \( M_D \) and \( \mu^{-1} \) is in \( M_D \) we get that \( \mu^{-1} \circ k_{G(v)} \) is in \( M_D \). By HLRP condition 7d we get that \( \mu^{-1} \circ k_{G(v)} \) is an isomorphism from which we get that \( k_{G(v)} \) is an isomorphism too. It implies that \( o_v \) is an isomorphism. That is, \( f_B(v) \cong f_D(o_G(v)) \).

Summing up, we get that for each \( v \in V_{m \rightarrow 1}(A_g) \)

- If \( v = l_G(g_G(w)) \) then the object \( f_D(v) \) with arrows \( l_{G_G}(w) \) and \( k_{f_G(a)} \) is the pushout of \( f_C(g_G(w)) \) \( \cong f_A(w) \xrightarrow{k} f_B(f_G(w)) \).
- If \( v = l_G(w) \) and \( w \notin g_G(V_{G_A}) \) then we have that \( l_w \) is an isomorphism
- If \( v = o_G(w) \) and \( w \notin f_G(V_{G_A}) \) then we have that \( o_w \) is an isomorphism

From this and the fact that (3) is a pushout, by applying Theorem A.2.1, we get that (1) is a pushout. But the fact that (1) + (2) and (1) are pushouts implies that (2) is a pullout too.

5. Consider the following pushout diagram

\[
\begin{array}{ccc}
  A & \xrightarrow{f} & B \\
  \downarrow g & & \downarrow k \\
  C & \xrightarrow{h} & D
\end{array}
\]

Assume that \( g, f, k, h \) are all in \( M \). Take an arbitrary morphism \( z : N \to D \). Consider the following diagrams where diagrams (3), (4) and (5) are
pullbacks:

\[(G_{N_C}, m_{N_C}) \xrightarrow{i_G^C} (G_C, m_C)\]

\[\downarrow f_G^C \hspace{2cm} \downarrow h_G \hspace{2cm} (3) \]

\[(G_N, m_N) \xrightarrow{z_G} (G_D, m_D)\]

\[(G_{N_B}, m_{N_B}) \xrightarrow{i_G^B} (G_B, m_B)\]

\[\downarrow f_G^B \hspace{2cm} \downarrow k_G \hspace{2cm} (4) \]

\[(G_N, m_N) \xrightarrow{z_G} (G_D, m_D)\]

\[(G_{N_A}, m_{N_A}) \xrightarrow{i_G^A} (G_A, m_A)\]

\[\downarrow f_G^A \hspace{2cm} \downarrow h_G \circ g_G \hspace{2cm} (5) \]

\[(G_N, m_N) \xrightarrow{z_G} (G_D, m_D)\]

Then the following diagram is a pushout

\[(G_{N_A}, m_{N_A}) \xrightarrow{u_{f_G}} (G_{N_B}, m_{N_B})\]

\[\downarrow u_{g_G} \hspace{2cm} \downarrow p_G \hspace{2cm} (6) \]

\[(G_{N_C}, m_{N_C}) \xrightarrow{\alpha_G} (G_N, m_{N_B})\]

Arrows \(u_{g_G}, u_{f_G}\) are unique arrows induced by arrows \(g_G \circ i_G^A\) and \(f_G \circ i_G^A\).

For all \(v \in V_{m_{N_C}^{-1}(A_B)}\) either \(i_G^C(v) = g_G(w)\) for a certain \(w \in V_{m_{N_B}^{-1}(A_B)}\) or \(i_G^C(v) \notin g_G(V_{G_A})\). In the former case we have that the following diagram is a pushout with all arrows in \(M_D\)

\[f_A(w) \xrightarrow{f_w} f_B(f_G(w))\]

\[g_w \downarrow \hspace{2cm} k_{f_G(w)} \downarrow\]

\[f_C(g_G(w)) \xrightarrow{h_G(w)} f_D(h_G(g_G(w)))\]

Besides, \(z_G(j_G^G(v)) = h_G(i_G^C(v)) = h_G(g_G(w))\) which implies that \(z_G(v) : f_N(j_G^G(v)) \rightarrow f_D(h_G(g_G(w)))\) from which it follows that there exists the
pullback of the diagram $f_C(g_G(w)) \xrightarrow{h_{g_G}(w)} f_D(h_G(g_G(w))) \xrightarrow{z_{j_G}^{C(v)}} f_N(j_G(v)).$

Denote that pullback and the corresponding arrows by $pb_{N_C}(w)$ and $i_C^v$. In the case of $i_G^v(v) \not\in g_G(V_{G_A})$ we have that $h_{i_G^v(v)} = id_{f_C(j_G^v(v))}$. But $f_N(j_G^v(v))$ with arrows $z_{j_G^v}^{C(v)}$ and $id_{f_N(j_G^v(v))}$ is the pullback of

$$f_C(i_G^v(v)) \xrightarrow{id_{f_C(j_G^v(v))}} f_D(h_G(i_G^v(v))) \xrightarrow{z_{j_G}^{C(v)}} f_N(j_G^v(v))$$

By Theorem A.2.2 we get that the object $N_C = (G_{N_C}, m_{N_C}, f_{N_C})$ where

$$f_{N_C}(v) = \begin{cases} pb_{N_C}(w) & \text{if } i_C^v(v) = g_G(w) \\ f_N(j_G^v(v)) & \text{otherwise} \end{cases}$$

with morphisms $i_C = (i_G^v, \{i_C^v|v \in V_{m_{N_C}}^{-1}(Ag) \land i_G^v(v) = g_G(w)\} \cup \{z_{i_G^v}^{C(v)}|i_C^v(v) \not\in g_G(V_{G_A}) \land v \in V_{m_{N_C}}^{-1}(Ag)\})$ and $j_C = (j_G^v, \{j_C^v|v \in V_{m_{N_C}}^{-1}(Ag) \land j_G^v(v) = g_G(w)\} \cup \{id_{f_N(j_G^v(v))}|i_C^v(v) \not\in g_G(V_{G_A}) \land v \in V_{m_{N_C}}^{-1}(Ag)\}$ is the pullback of $C \xrightarrow{h} D \xrightarrow{f} N$. The existence of pullback objects $N_B$ and $N_A$ can be proven in the similar manner. That is, the pullback of $B \xrightarrow{i} D \xrightarrow{f} N$ is the object $N_B = (G_{N_B}, m_{N_B}, f_{N_B})$ with morphisms $i_B = (i_G^v, \{i_B^v|v \in V_{m_{N_B}}^{-1}(Ag) \land i_G^v(v) = f_G(w)\} \cup \{z_{i_G^v}^{B(v)}|i_B^v(v) \not\in g_G(V_{G_A}) \land v \in V_{m_{N_B}}^{-1}(Ag)\})$ and $j_B = (j_G^v, \{j_B^v|v \in V_{m_{N_B}}^{-1}(Ag) \land j_G^v(v) = g_G(w)\} \cup \{id_{f_N(j_G^v(v))}|i_B^v(v) \not\in g_G(V_{G_A}) \land v \in V_{m_{N_B}}^{-1}(Ag)\})$ is the pullback of $A \xrightarrow{k} D \xrightarrow{f} N$. The pullback of $A \xrightarrow{k} D \xrightarrow{f} N$ is the object $N_A = (G_{N_A}, m_{N_A}, f_{N_A})$ with morphisms $i_A = (i_G^v, \{i_A^v|v \in V_{m_{N_A}}^{-1}(Ag) \land w = i_G^v(v)\})$ and $j_A = (j_G^v, \{j_A^v|v \in V_{m_{N_A}}^{-1}(Ag) \land w = j_G^v(v)\})$. Consider the unique arrows $u_f$ and $u_g$ induced by arrows $f \circ i_A$ and $g \circ i_A$, i.e.

$$i_B \circ u_f = f \circ i_A, \quad i_C \circ u_g = g \circ i_A, \quad j_B \circ u_f = j_A \circ u_g.$$

By the proof of Theorem A.2.2 we get that $u_f = (u_{f_G}, \{u_{f_v}|v \in V_{m_{N_A}}^{-1}(Ag) \land w = i_G^v(v)\})$ and $u_g = (u_{g_G}, \{u_{g_v}|v \in V_{m_{N_A}}^{-1}(Ag) \land w = i_G^v(v)\})$ where $u_{f_w}$ and $u_{g_w}$ are the unique arrows induced by $f_w \circ i_A^v$ and $g_w \circ i_A^v$, i.e.

$$i_B \circ u_{f_w} = f_w \circ i_A^v, \quad i_C \circ u_{g_w} = g_w \circ i_A^v, \quad j_B \circ u_{f_w} = j_A \circ u_{g_w}.$$

Then the following diagram is a pushout for all $v \in V_{m_{N_A}}^{-1}(Ag)$ such that $w = i_G^v(v)$.

\[
\begin{array}{ccc}
  f_{N_A}(v) & \xrightarrow{u_{f_v}} & f_{N_B}(u_{f_G}(v)) \\
  u_{g_v} \downarrow & & \downarrow p_v \\
  f_{N_C}(u_{g_{f_G}(v)}) & \xrightarrow{\alpha_v} & f_N(j_G^v(v))
\end{array}
\]

Note that $j_G^v(v) = j_G^v(u_{g_G}(v)) = j_G^v((i_G^v(v))^{1-1}(g_G(w)))$. It is also true that $z_{j_G^v}^v(v)$ is the morphism induced by morphisms $k_{f_G}(w) \circ i_A^v$ and $h_{g_G}(w) \circ i_A^v$. 
From this by using the proof of the Theorem A.2.2 we get that $N$ with arrows $o = \{o_v | v \in V_{m_A^{-1}(A_B)} \} \cup \{id_{f_N}(v) | v \notin u_{y_A}(V_{G_A})\}$ and $p = \{p_v | v \in V_{m_A^{-1}(A_B)} \} \cup \{id_{f_N}(v) | v \notin u_{y_A}(V_{G_A})\}$ is indeed a pushout. Besides the arrow induced by $k \circ i^B$ and $h \circ i^C$ coincides with $z$.

6. Consider the following diagram

\[
\begin{array}{ccc}
K & \xrightarrow{r} & A \\
| & | & | \\
(3) & p & g \\
| & | & | \\
A & \xrightarrow{f} & B \xleftarrow{l} (1) \\n| & | & | \\
E & \xrightarrow{k} & D \xleftarrow{m} (2) \\
| & | & | \\
& \xleftarrow{t} & Z \xrightarrow{z} (4) \\
\end{array}
\]

Assume that (1),(2) are pushouts, (3) and (4) are pullbacks, and all arrows of the diagram are in $M$. Then diagrams $(1_G),(2_G)$ are pushouts and $(3_G),(4_G)$ are pullbacks:

\[
\begin{array}{ccc}
(G_K,m_K) & \xrightarrow{r_G} & (G_A,m_A) \xrightarrow{g_G} (G_B,m_B) \xrightarrow{n_G} (G_B,m_B) \\
| & | & | \\
(3_G) & p_G & C \xleftarrow{n_G} C \xleftarrow{n_G} \\
| & | & | \\
| & | & | \\
(1_G) & l_G & (2_G) \xrightarrow{\alpha_G} \alpha_G \\
| & | & | \\
(1_G) & l_G & (2_G) \xrightarrow{\alpha_G} \alpha_G \\
| & | & | \\
(G_E,m_E) & \xrightarrow{k_G} (G_D,m_D) \xleftarrow{m_G} (G_F,m_F) \\
| & | & | \\
| & | & | \\
| & | & | \\
(G_Z,m_Z) & \xrightarrow{t_G} (4_G) \xrightarrow{z_G} Z \xrightarrow{z_G} \\
\end{array}
\]

Since HLRP conditions hold for typed graphs, we get that diagrams $(5_G)$
and (6G) are pushouts.

\[
(G_A, m_A) \xrightarrow{r_G} (G_K, m_K) \xrightarrow{p_G} (G_C, m_C)
\]

\[
G_E, m_E \xleftrightarrow{t_G} G_Z, m_Z \xrightarrow{z_G} G_F, m_F
\]

For all \( w \in V_{m^{-1}_w(A)} \) three cases can be distinguished:

**There exists such a \( v \) that** \( f_G \circ r_G(v) = w \)

Then we get the following diagram

\[
\begin{align*}
& f_K(v) \\
& \downarrow r_v & \downarrow (3_v) & \downarrow p_v \\
& f_A(r_G(v)) \xrightarrow{g_{r_G(v)}} f_B(g_B(r_G(v))) \xrightarrow{n_{p_G(v)}} f_C(r_G(v)) \\
& f_{r_G(v)} \downarrow (1_v) & \downarrow l_{g_B(r_G(v))} & \downarrow (2_v) & \downarrow o_{r_G(v)} \\
& f_E(w) \xrightarrow{k_w} f_D(k_G(w)) \xrightarrow{m_{p_G(v)}} f_F(o_G(p_G(v))) \\
& \downarrow t_q & \downarrow (4_v) & \downarrow z_q \\
& f_Z(q) \\
\end{align*}
\]

where \( t_G(q) = w \). According to Theorem A.2.1 and Theorem A.2.2 (1_v), (2_v) are pushouts and (3_v) and (4_v) are pullbacks. Since \( D \) satisfies HLRP conditions we get that the diagram (5_v) below is a pushout.

\[
\begin{align*}
& f_A(r_G(v)) \xrightarrow{r_v} f_K(v) \\
& f_G(r_G(v)) \downarrow (5_v) & \downarrow u_v \\
& f_E(w) \xrightarrow{t_q} f_Z(q)
\end{align*}
\]

Here \( u_v \) is the unique arrow induced by arrows \( f_{r_G(v)} \circ r_v, o_{r_G(v)} \circ p_v \) according to the pullback property of \( f_Z(q) \).
\( w \notin f_G(V_{\mathcal{G}_A}) \) There exists an \( q \in V_Z \) such that \( t_G(q) = z \). Indeed, by the construction of pushouts \( k_G(w) \notin f_G(V_{\mathcal{G}_A}) \) which implies that there exists such a \( v' \in V_{\mathcal{G}_P} \) that \( m_G(v') = k_G(w) \). That is, \( q = (w, v') \). Besides by the Theorem A.2.1 in this case \( k_w = \text{id}_{f_P(w)} \) and \( m_{v'} = \text{id}_{f_P(v')} \). This implies by Theorem A.2.2 that \( f_Z(q) = f_E(w) \) and \( t_q = \text{id}_{f_P(w)} \). Indeed, \( f_Z(q) \) with \( t_q \) and \( z_q \) is supposed to be the pullback \( f_E(w) \xrightarrow{id_{f_P(w)}} f_D(k_G(w)) \xleftarrow{id_{f_P(v')}} f_E(v') \). But \( f_E(w) \) with \( id_{f_P(w)} \) is the pullback of the diagram above, so by the uniqueness of pullbacks we get \( f_Z(q) \cong f_E(w) \) and \( t_q \) and \( z_q \) are isomorphisms. That is, we can take \( f_Z(q) = f_E(w) \) and \( t_q = z_q = id_{f_P(w)} \).

\( \exists v \in V_{\mathcal{G}_A} : w = f_G(v) \land v \notin r_G(V_{\mathcal{G}_A}) \) In this case \( k_G(w) \notin \ell(V_{\mathcal{G}_A}) \setminus m_G(V_{\mathcal{G}_P}) \).

That is, \( w \notin t_G(V_{\mathcal{G}_A}) \). But since \( k_G(w) \notin m_G(V_{\mathcal{G}_P}) \) from the fact that (2) is a pushout by Theorem A.2.1 we get that \( f_D(k_G(w)) = f_B(g_G(v)) \) and \( l_{g_G(v)} = \text{id}_{f_P(g_G(v))} \). Again, from the fact that (1) is a pushout by Theorem A.2.1 we get that the following diagram is a pushout:

\[
\begin{array}{ccc}
  f_A(v) & \xrightarrow{g_v} & f_B(g_G(v)) \\
  \downarrow f_v & & \downarrow \text{id}_{f_P(g_G(v))} \\
  f_E(w) & \xrightarrow{k_w} & f_B(g_G(v))
\end{array}
\]

But since \( f_v, g_v, k_w \) and \( \text{id}_{f_P(g_G(v))} \) are in \( M_D \), the diagram (5a) is also a pullback. But \( f_E(w) \) with \( k_w \) and \( \text{id}_{f_P(w)} \) is the pullback of \( f_E(w) \xrightarrow{k_w} f_B(g_G(v)) \xleftarrow{\text{id}_{f_P(v')}} f_B(g_G(v)) \). By the uniqueness of pullbacks we get \( f_E(w) \cong f_A(v) \) and \( f_v \) is an isomorphism, that is, we can take \( f_A(v) = f_E(w) \) and \( f_v = \text{id}_{f_A(v)} \). Note that \( w \notin t_G(V_{\mathcal{G}_A}) \).

To sum up, we get that \( f_E(w) \) is either the pushout object of diagram (5a) \( (f_G(r_G(v)) = w) \) or \( f_E(w) = f_A(v) \) and \( f_v = \text{id}_{f_A(v)} (w = f_G(v) \land v \notin r_G(V_{\mathcal{G}_A})) \) or \( f_E(w) = f_Z(q) \) and \( t_q = \text{id}_{f_P(w)} (w = t_G(q) \land w \notin f_G(V_{\mathcal{G}_A})) \).

From the discussion above by using Theorem A.2.1 we get that (5) is indeed a pushout. The fact that the diagram (6) is a pushout can be done in the similar way.

```
A ---- K ---- C
|     \   |     /
|      \  |     /
|       \ u-- (6) o
|         |
E ------ Z ------ F
```

where \( u = (u_G, \{ u_v \in V_{m_G^{-1}A} \}) \) is the unique arrow induced by arrows \( f \) and \( o \) according to the pullback property of \( Z \).
7. It is easy to see that $M_D$ contains all identity morphisms and isomorphisms and closed under composition then so does $M$ constructed according to Theorem 5.2.2. It is rather obvious that if all elements of $M_D$ are monomorphisms in $D$ then all elements of $M$ constructed according to Theorem 5.2.2 will be monomorphisms too.

Let $f : A \to B$ be an arrow from $M$. Let $g : B \to A$ be an arrow such that $f \circ g = id_A$. Then it means that $f_G \circ g_G = id_A$ and for all $v \in V^{-1}_{m-1}(A_g)$ $f_g(v) \circ g_v = id_{f_G(v)}$. Since $f_g(v)$ is in $M_D$ we get that $f_g(v)$ is an isomorphism. But from $f_G \circ g_G = id_A$ we get that $g_G$ and $f_G$ are isomorphisms which implies that $g_G$ is surjective. Thus we get that $f_w$ is an isomorphism for each $w \in V^{-1}_{m}(A_g)$. That means that $f = (f_G, \{ f_w | w \in V^{-1}_{m}(A_g) \})$ is an isomorphism too.

Consider the following diagram where all arrows are in $M$ and (1) is a pushout and (2) is a pullback.

\[
\begin{array}{ccc}
C & \xrightarrow{f} & A \\
\downarrow{g} & & \downarrow{(1) \ p} \\
B & \xrightarrow{r} & D
\end{array}
\quad
\begin{array}{ccc}
C & \xrightarrow{f} & A \\
\downarrow{g} & & \downarrow{(2) \ k} \\
B & \xrightarrow{l} & Z
\end{array}
\]

According to the proof of the Theorem A.2.1 the morphism $u : D \to Z$ induced $l$ and $k$ has the following shape $u = (u_G, \{ u_v | v \in V^{-1}_{m}(A_g) \})$ where

\[
u_v = \begin{cases} 
  l_w & \text{if } w \notin g_G(V_G) \land r_G(w) = v \\
  k_w & \text{if } w \notin f_G(V_G) \land p_G(w) = v \\
  u_w & \text{if } p_G(f_G(w)) = v 
\end{cases}
\]

where $u_w : f_D(v) \to f_Z(u_G(g_G(w)))$ is the unique morphism induced by $l_{g_G(w)}$ and $k_{f_G(w)}$. It is clear that $l_w$ in the first branch of the definition of $u_v$ and $k_w$ in the second branch of the definition of $u_v$ are both in $M_D$. So, if we prove that $u_w$ is in $M_D$ then we get that $u_v$ is in $M$ for each $v \in V^{-1}_{m}(A_g)$. Note that the diagram below is a pushout.

\[
\begin{array}{ccc}
f_C(w) & \xrightarrow{f_w} & f_A(f_G(w)) \\
\downarrow{g_w} & & \downarrow{(1_v) \ p_{f_G(w)}} \\
f_B(g_G(w)) & \xrightarrow{f_g(w)} & f_D(v)
\end{array}
\]

Notice that all arrows of the diagram $(1_v)$ are in $M_D$. On the other hand
by the Theorem A.2.2 we get that the diagram (2_v) below is a pullback.

\[
\begin{array}{ccc}
  f_C(w) & \xrightarrow{f_w} & f_A(f_G(w)) \\
  g_w & & (2_v) \\
  f_B(g_G(w)) & \xrightarrow{f_{w_G}} & f_Z(l_G(g_G(w)))
\end{array}
\]

Note that all arrows of the diagram (2_v) are in \( M_D \). By the assumption about the category \( D \) we get that \( u_w \) is indeed in \( M_D \). That is, whatever \( v \in V_{m^{-1}_A(B)} \) we take, \( u_v \) is always in \( M_D \). That is, \( u \in M \).

8. Consider \( h : A \to B \). Then the pullback of \((G_A, m_A) \xrightarrow{h_A} (G_B, m_B) \xrightarrow{h_B} (G_A, m_A)\) exists (the category of typed graphs is finitely complete). Denote the pullback object by \((G_C, m_C)\) and the appropriate morphisms by \( f_G \) and \( g_G \). Because of the construction of the pullback in the category of typed graphs with injective morphisms we get that \( f_G = g_G = id_{G_A} \) and \((G_C, m_C) = (G_A, m_A)\). For each \( v \in V_{m^{-1}_A(B)} \) the pullback of \( f_A(v) \xrightarrow{h_A} f_B(h_G(v)) \xrightarrow{h_B} f_A(v) \) exists. Denote this pullback by \( p_{v_e} \) and the pullback morphisms by \( f_{p_{v_e}} \) and \( g_{p_{v_e}} \). Then it is easy to see that by Theorem A.2.2 object \( C = (G_A, m_A, f_C) \) where \( f_C(w) = p_{w_G} (w \in V_{m^{-1}_A(B)}), \) together with morphisms \( f = (id_{G_A}, \{f_{p_{v_e}} | v \in V_{m^{-1}_A(B)}\}) \) and \( g = (id_{G_A}, \{g_{p_{v_e}} | v \in V_{m^{-1}_A(B)}\}) \) is the pullback of \( A \xrightarrow{h_A} B \xleftarrow{h_B} A \) We get that \((G_A, m_A)\) with arrows \( id_{G_A}\) is the pushout of \((G_A, m_A) \xrightarrow{id_{G_A}} (G_A, m_A) \xrightarrow{id_{G_A}} (G_A, m_A)\). But for each \( w \in V_{m^{-1}_A(B)} \) we get that the pushout of \( f_A(w) \xrightarrow{p_{w}} f_C(w) \xrightarrow{f_{p_{w}}} f_A(w) \) exists. Let the object \( p_{w_G} \) with morphisms \( o_{w} \) and \( p_{w_G} \) be the pushout of \( f_A(w) \xrightarrow{p_{w}} f_C(w) \xrightarrow{f_{p_{w}}} f_A(w) \). That is, the diagram below is a pushout.

\[
\begin{array}{ccc}
  f_C(w) & \xrightarrow{f_{p_{w}}} & f_A(w) \\
  g_{p_{w}} & & (2_v) \\
  f_A(w) & \xrightarrow{o_{w}} & p_{w_G}
\end{array}
\]

Then by Theorem A.2.1 the object \( \tilde{A} = (G_A, m_A, f_{\tilde{A}}) \) where \( f_{\tilde{A}}(w) = p_{w_G} \) with morphisms \( p = (o_G, \{p_{w_G} | v \in V_{m^{-1}_A(B)}\}) \) and \( o = (o_G, \{o_{w} | v \in V_{m^{-1}_A(B)}\}) \) is the pushout of \( A \xrightarrow{o_G} C \xleftarrow{f_{\tilde{A}}} A \). Besides, the unique arrow \( u_k : \tilde{A} \to B \) is in \( M \). Indeed, according to Theorem A.2.1 we get that \( u_k \) is of the form \( u_k = (u_G, \{u_{v_k} | v \in V_{m^{-1}_A(B)}\}) \) where \( u_v : f_{\tilde{A}}(v) \to f_B(u_G(v)) \)
is the unique arrow induced by arrows \( h_v \), therefore \( u_s \) belongs to \( M_D \). This means that \( u_h \) belongs to \( M \).

9. For typed graphs with injective morphisms the pushout complement is unique. That is, if \( f : A \to B \) is in \( M \) and there exists an object \( C \) and arrows \( o \) and \( k \) such that the following diagram is a pushout.

\[
\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{h} & & \downarrow{o} \\
C & \xrightarrow{k} & D
\end{array}
\]

then the diagram below will be a pushout too

\[
\begin{array}{c}
(G_A, m_A) \xrightarrow{f_G} (G_B, m_B) \\
h_G \downarrow & & \downarrow o_G \\
(G_C, m_C) & \xrightarrow{k_G} & (G_D, m_D)
\end{array}
\]

and \( (G_C, m_C), h_G, k_G \) are unique up to isomorphism. But for all \( w \in V_{m_A^{-1}(A_B)} \) the following diagram is a pushout

\[
\begin{array}{ccc}
f_A(w) & \xrightarrow{f_w} & f_B(f_G(w)) \\
h_w \downarrow & & \downarrow o_{f_G(w)} \\
f_C(h_G(w)) & \xrightarrow{k_{h_G(w)}} & f_D(o_G(f_G(w)))
\end{array}
\]

Since \( f \) in \( M \) we have that \( f_w \) is in \( M_D \). From the assumption that \( D \) satisfies HLRP conditions we get that \( f_C(h_G(w)) \) with \( h_w \) and \( k_{h_G(w)} \) is unique up to isomorphisms. Besides, for all \( v \in V_{m_C^{-1}(A_B)} \) such that \( v \not\in h_G(V_{G_A}) \) we have that because of the colimit construction from Theorem A.2.1 \( f_C(v) = f_D(k_G(v)) \) and \( k_v = id_{f_C(v)} \).

That is, we have that \( (G_C, m_C) \) with arrows \( h_G \) and \( k_G \) is unique up to isomorphisms and that for all \( w \in V_{m_A^{-1}(A_B)} \) the pushout complement object \( f_C(h_G(w)) \) with \( h_w \) and \( k_{h_G(w)} \) is unique up to isomorphism and that \( f_C(v) = f_D(k_G(v)) \) and \( k_v = id_{f_C(v)} \) where \( v \in V_{m_C^{-1}(A_B)} \setminus h_G(V_{G_A}) \). It means that \( C \) with arrows \( h \) and \( k \) is itself unique up to isomorphism. Indeed, if we take an object \( C' \) and arrows \( h' \) and \( k' \) such that the diagram
then we get that $(G_{C'}, m_{C'}) \cong (G_C, m_C)$ and $h_G = \phi_G \circ h'_G$ and $k_G = k'_G \circ \phi^{-1}_G$ where $\phi : (G_{C'}, m_{C'}) \rightarrow (G_C, m_C)$. For all $w \in V_{m^{-1}_A(A_g)}$ we have that $f_C(h_G(w)) \cong f_C'(h_G(w))$ with isomorphism $\phi_{h'_G(w)} : f_C'(h_G(w)) \rightarrow f_C(h_G(w))$ and $h_w = \phi_{h'_G(w)} \circ h'_w$ and $k_w = k'_w \circ \phi^{-1}_G(w)$. For all $v \in V_{m^{-1}_C(A_g)} \setminus h_G(V_{G_A})$ we have that $f_C(v) = f_D(k'_G(v)) = f_D(k_G \circ \phi_G(v)) = f_C(\phi(v))$ and $k'_v = k_{\phi_G(v)} = \text{id}_{f_C(\phi_G(v))}$. That is, $\phi = (\phi_G, \{\phi_v|v = h_G(w)\} \cup \{\text{id}_{f_C(v)}|v \in V_{m^{-1}_C(A_g)} \setminus h_G(V_{G_A})\}) : C' \rightarrow C$ is an isomorphism such that $h = h'_G \circ \phi$ and $k = k'_G \circ \phi^{-1}$. 

\[
\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{h'} & & \downarrow{o} \\
C & \xrightarrow{k'} & D
\end{array}
\]
Bibliography


