

Graph-Transformational Swarms

A Graph-Transformational Approach to Swarm Computation

von
Larbi Abdenebaoui

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

Gutachter:
Prof. Dr. Hans-Jörg Kreowski
Prof. Dr. Jürgen Pannek

Vorgelegt im Fachbereich 3 (Mathematik/Informatik)
der Universität Bremen
im August 2016

Acknowledgements

This thesis would not have been possible without the guidance, the help and the support of many people, to whom I would like to express my gratitude.

First, I would like to thank my supervisor, Hans-Jörg Kreowski, for his engaged and continuous support and advising for my PhD study and related research, for sharing his knowledge and expertise, and for given me the freedom to pursue research of my own interest but also to be there whenever I asked for. I would like also to thank him for affording me the opportunity to work in the friendly and inspiring atmosphere of his research group.

I would like to thank Jürgen Pannek for offering me the chance to present my work in his research group, for his comments and remarks and for accepting, in a short time, to referee my thesis.

My sincere thanks also goes to Sabine Kuske for her valuable help and advices. I am grateful for all the discussions and the support in different aspects. It was a pleasure to make research and write papers with her and Hans-Jörg Kreowski. I learn a lot from them.

Thank you Marcus Ermler for sharing the office room with me and for all comments and advices concerning my PhD. I would like to thank my other colleagues: Melanie Luderer and Caroline von Totth, for the agreeable time in the theoretical computer science group. In general, I have enjoyed a friendly, supportive, and open exchange with all my colleagues.

I would like to thank my friend Robert Wogatzke for his proofreading, comments and motivations. My friend Bertold Bongardt has also proposed several comments and suggestions that I received with gratitude.

I gratefully acknowledge the financial support which I have received from the Rosa Luxemburg Foundation in form of a scholarship. It was a pleasure to participate to different workshop that the foundation has organized. The required reports were useful to structure my research work in order to achieve the thesis within the regular time. In this context, I would like to thank the International Graduate School for Dynamics in Logistics for the different courses as well as for all other PhD-related offers. Particularly, I would like to thank Ingrid Rügge for her engagement.

Lust but not least, I would like to thank my family and friends for their patience and support. Particularly, my wife Elif Gökpinar and our child Mina Jiyan have support me unconditionally. Your support, encouragement and love have been and are a source for inspiration and a means for continuation. A considerable thank also goes to the family Gökpinar for their continuous support. I would like to thank my family in Morocco, that support me so much during my entire life. For that, I switch to Arabic (in Morocco speaked version, called also Darija):

شكرا إلى عائلتي الصغيرة في المغرب، في إسبانيا و حديثا في فرنسا. شكراً بزاف خويا مصطفى، خويا عبدالرحيم، خويا بوشعيب وأختي إلهام. وبطبيعة الحال شكراً للميمة لي ربت وكبرت وسهرت... أهدي هذه الدكتوراة إلى روح الفقيد بابا الله يرحمو.

Contents

1	Introduction	1
1.1	Objectives	4
1.2	Related work	4
1.3	Structure of the thesis	5
2	Swarms and swarm computing	9
2.1	Swarms in nature	9
2.2	Major swarm computing methods	16
2.3	Other methods	25
2.4	Summary	27
3	Graph transformation	29
3.1	General ideas and backgrounds	29
3.2	Preliminaries	30
3.3	Graphs	31
3.4	Rules and their applications	35
3.5	Application context	41
3.6	Parallel rule application	43
3.7	Graph grammars and graph transformation units	49
3.8	Graph transformation tools	54
3.9	First implementations	60
3.10	Summary	63
4	Graph-transformational swarms	65
4.1	The main ideas of swarms	65
4.2	Graph-transformational swarms and their computations	73
4.3	Examples	75
4.4	Stochastic control	80
4.5	Modeling: practical considerations	86
4.6	Summary	96
5	Unification capability	97
5.1	Preliminaries	97
5.2	Graph-transformational particle swarm	98

5.3	Graph-transformational cellular automata	103
5.4	Ant colony optimization as a framework	106
5.5	Graph-transformational ant colony	113
5.6	Summary	120
6	Swarms with stationary members	123
6.1	Cloud computing	123
6.2	Stationary members	124
6.3	Examples	125
6.4	Summary	128
7	Modeling in dynamic logistic networks	129
7.1	From swarms in nature to logistic networks	129
7.2	Routing of automated guided vehicles	132
7.3	Summary	141
8	Conclusion	143
8.1	Summary	143
8.2	Contributions	145
8.3	Outlook	146

Chapter 1

Introduction

Computer systems are becoming increasingly distributed and interconnected. Various emerging notions, such as smart grids, system of systems, industry 4.0 or cyber-physical systems have gained more and more importance during the last few years. All of them propose to solve engineering problems by using several autonomous components that act in parallel and are interconnected, foremost using Internet technologies. These emerging concepts look very promising, but also exhibit various technical challenges. For instance, how is it possible to develop decentralized control mechanisms that produce a desired emerging behavior to solve a given task or how to model such solutions in order to analyze their behavior in terms of complexity and correctness? These are two major questions that this thesis attempts to answer. Indeed, it provides graph-transformational swarms as a novel concept that combines the ideas and principles of swarms and swarm computing and the formal methods of graph transformation to model distributed systems. Figure 1.1 illustrates how graph-transformational swarms captures the advantages of swarms and swarm computing and of graph transformation. This combination is introduced in the following three paragraphs in more detail.

Swarms and swarm computing Swarms in nature are fascinating phenomena where animals living in groups cooperate with each other to solve complex problems. Many familiar examples can illustrate the teamwork within a swarm. For instance, both ant colonies and bee hives build nests and manage the resources inside of them. Furthermore, they forage for food and transport it in an efficient and flexible way, drawing upon elements of mutuality. Schools of fishes and flocks of birds migrate every year traversing very long distances. They act as one body in the face of predator attacks, overcoming complex hydrodynamic constraints. In addition to the macroscopic scale, several organisms on the microscopic scale seem to act like a swarm if one considers their cellular components as swarm members. For example, microbial colonies, nervous systems, or immune systems.

Several studies agree on the assumption that the complex behavior of swarms arises from relatively simple rules on the individual level (see e.g., [9, 74, 18, 45, 16, 38]). In biology, the underlying mechanism is also known as *self-organization*: The individuals that constitute the group interact locally with other group members without further

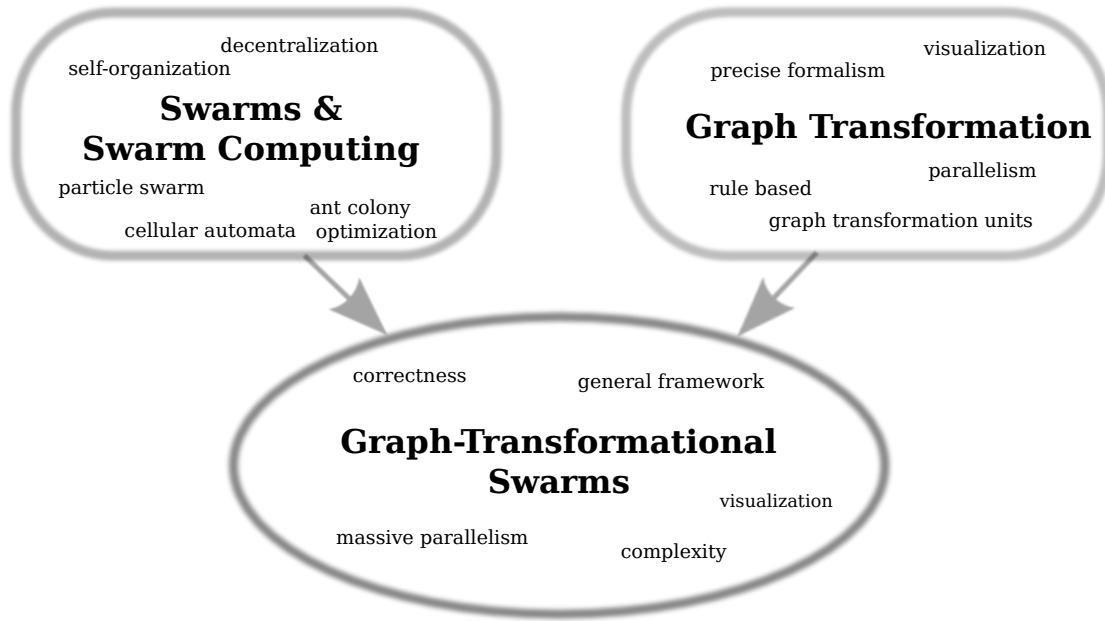


Figure 1.1: An overview of the main components in the thesis and their advantages.

knowledge of the global behavior of the entire group. Furthermore, all members play the same role without any hierarchical structure.

The robustness and efficiency of solving complex problems based on the simplicity of the individual level are characteristics that make swarms promising models for solving problems in engineering domains. Particularly, in computer science one encounters quite a variety of swarm concepts and swarm algorithms in the literature (see, e.g., [6, 7, 33, 50, 71]). Moreover, there are several general computational approaches which are subsumed under the heading of swarm intelligence. But there seems to be no common framework covering all approaches to swarms.

Throughout this dissertation, the term “swarms and swarm computing” is used to refer to swarms in nature and the approaches in computer science inspired thereby. In contrast to the term “swarm intelligence”, which is very often employed to designate both – the specific animal behavior in nature as well as the inspired methods in different engineering fields – the chosen term defines computer science more precisely as an application field. Moreover, it emphasizes the difference between the biological foundations and the approaches inspired by them avoiding ambiguities.

Graph transformation In computer science and in many other branches of science, graphs are well-established to represent information. They can visualize structures consisting of entities and the relations between them. Such structures can correspond in turn to certain states of a dynamic system such as the initial state or the goal state. Graph transformation extends the static representation capability of graphs by offering the possibility to specify and visualize the dynamics of a system. The main idea therein is the transformation of graphs based on the application of rules. A rule, in this context,

defines what should be deleted and/or added in a graph.

Graph transformation was introduced in the late sixties and early seventies [78, 84, 91, 29]. Since then, the field has continued to evolve, leading to several theoretically well-founded approaches and providing various applications in different branches of computer science. This development is documented in three handbooks [89, 25, 90, 32] as well as in several other articles and monographs (see, e.g., [26, 31, 23, 28, 53, 14, 27]). A graph transformation approach defines – amongst other things – what the rules look like exactly and how they are applied to graphs to perform transformations. This thesis chooses an approach of graph transformation in such a way that rules can be applied in parallel and that their parallel applicability follows from the applicability of each of the involved rules and additional conditions.

In order to structure the rules and control their application, the concept of graph transformation units [55] is used. A graph transformation unit consists of a set of rules, a control condition that regulates the application of rules as well as descriptions of initial and terminal graphs. This thesis investigates the capabilities of graph transformation units to specify swarm members and how to generate similar members that have the same role in a swarm.

Graph-transformational swarms This thesis proposes graph-transformational swarms as a general approach to swarm computation. It is designed to unify different existing concepts of swarm computing, motivated by the lack of a common approach in the literature.

A graph-transformational swarm consists of an arbitrary number of members. The members can have the same or different role in the swarm. They act simultaneously within a common environment which is represented as a graph. Moreover, there may be a co-operation condition to regulate the interaction and cooperation of the members as well as a goal to be reached. Members are modeled as graph transformation units which are computational devices based on rules. The key is that the framework of graph transformation provides the concept of parallel rule application to formalize the simultaneous actions of swarm members.

The employment of graph transformation to formulate the components needed in graph-transformational swarms together with the unification approach yield to the following features and advantages:

- different models of swarm computation can be compared with each other more easily within a common framework,
- transferring the results of one model to other models is made simple,
- new methods that combine the ideas of different swarm computing approaches can be comfortably developed,
- a graphical specification can simplify the analysis and survey of complex systems,
- a well-defined mathematical semantics is used for formal specifications,

- a systematic specification of parallelism allows the development distributed models and
- graph transformation tools can be employed for simulation, model checking and SAT solving in a standardized way.

1.1 Objectives

In order to reach the aim of building a framework that exhibits the desired advantages, the following four objectives have been defined. The first objective of this thesis is to determine the common principles of swarms and swarm computing. A part of this objective is to review the problem-solving behavior of social animals and to report the main related theories from biology. Another part of this objective is to designate the major swarm computing approaches, specifying their shared characteristics. Consequently, this objective determines the requirements that a common framework has to satisfy.

The second objective concerns the definition of methods of graph transformation needed to respond to the stated requirements. More precisely, a suitable graph transformation approach has to be specified. Subsequently, its components have to be adapted and extended. Particularly the question, how suitable is graph transformation for modeling parallel processes has to be thoroughly discussed.

The third objective is based on the two previous: The design of a concept that unifies the major swarm computing approaches, that is in accordance with biological foundations and that is formulated using graph transformation capability. The syntax and the semantics of each proposed component has to be specified and illustrated. The computation process has also to be well specified in such a way that it is possible to describe the evolution of the system systematically.

The fourth and last objective is to evaluate the developed concept by means of several examples. In order to demonstrate the unification capability of the concept, the major swarm computing approaches have to be embedded in the framework. Furthermore, the advantages of the formal semantics and the visualization capabilities have to be explored using different case studies.

1.2 Related work

There is, to the author's knowledge, no other comparable research work in the literature that attempts to unify the different swarm computing approaches such as this thesis does. However, this thesis is related to other research work, in terms of being influenced by their concepts or sharing certain ideas and methods. The chapters and sections in this thesis are equipped with specific paragraphs providing the underlying references. This section gives a brief overview of additional and closely related works.

The idea to employ graph transformation to model swarm computing methods was initiated in the works [63] and [64] using the notion of communities of autonomous units. The proposed solutions in these two works focus on a unique swarm computing approach namely, ant colony optimization. In particular, the salesperson and the capacitated vehicle routing problems were successfully modeled. The findings of these works has motivated the research on graph-transformational swarms. Since, graph transformation units have also been employed as basic component for modeling the swam members. However, the concept of graph-transformational swarms shifts several features from the level of graph transformation units to the swarm level. It makes the graph transformation units as simple as possible and, in return, proposes a powerful structuring concept in a higher level.

Graph-transformational swarms are also related to [51] where graph transformation has been used for plant modeling. The proposed *relational growth grammars* can be considered as a combination of L-systems and graph grammars. As a consequence, relational growth grammars extend the strength of L-systems for plant modeling to represent even more structures and their dynamics in a natural way. The common aspects of relational growth grammars and graph-transformational swarms is that both employ graph transformation to represent knowledge from biology. Furthermore the parallelism plays an important role in both concepts. However, the application fields are different. While relational growth grammars is designed to model biological systems, graph-transformational swarms uses the principles from biology to solve engineering problems. As a consequence, the structuring concept in graph-transformational swarms is more adapted to swarm-inspired solutions.

Graph-transformational swarms are related to other graph transformation approaches to parallelism and distribution as they are surveyed in [90] (see particularly the contributions by Litovsky, Métivier and Sopena, by Janssens and by Taentzer et al.) These related approaches consider parallel and distributed computing on the level of rule application rather than on the level of units as in the case of swarms. One of the most closely related approaches to swarms with stationary members seems to be the graph relabeling systems (see, e.g., [69, 5]). In this realated approach all node labels are changed simultaneously in every step implying massive parallelism and stationarity with respect to nodes.

1.3 Structure of the thesis

The overall structure of the thesis takes the form of seven chapters, including this introduction.

Chapter 2 provides an overview on swarms and swarm computing. It starts by reviewing swarms in nature reporting seminal experiments that have influenced the field of swarm computing. Afterwards, the major approaches in swarm computing are introduced. They consists of the three approaches: ant colony optimization, particle swarm

optimization and cellular automata. To show the diversity of the field, two relatively recent approaches are also briefly introduced. Finally, the chapter discusses the common principles of the presented approaches.

Chapter 3 introduces graph transformation focusing on the basic elements as far as needed for this thesis. It presents the appropriate approach with respect to the requirement of swarms specifying the used graphs and rules. In order to allow more flexibility in modeling, the notion of application context is also integrated in the approach. Chapter 3 provides a very important theorem concerning parallel rule applications, namely the generalized parallelization theorem. Graph transformation units are introduced as the basic structuring concept that is used to represent members and kinds of a swarm. The chapter starts to explore the possibility of implementing the introduced components in different graph transformation tools. It compares different tools and presents the first results of implementations in the one selected for the purpose of this work.

Chapter 4 introduces the notion of graph transformation swarms. It starts by combining the main ideas from Chapter 3 and Chapter 2 in order to define the components needed. It thereby provides the main principles of swarms and swarm computing with a graph transformation perspective. Graph-transformational swarms and their computations are then formally introduced. To illustrate the concept, two examples are given: a very simple ant colony and a swarm that computes Hamiltonian cycles. The chapter extends the notion of control condition in order to specify stochastic processes. Finally, the chapter discusses practical considerations when to implement the components of graph-transformational swarms.

Chapter 5 provides graph-transformational versions of the three major swarm computing approaches: ant colony optimization, particle swarm optimization and cellular automata. In this way, it demonstrates the unifying capabilities of the designed framework. The three approaches are recalled in a formal manner in this chapter. Particle swarm optimization as well as cellular automata are recalled, based on the so called canonical versions. Ant colony optimization is recalled as a framework embedding several methods.

Chapter 6 introduces stationary members. Stationary members are assigned to particular subgraphs of the considered environment graphs and are responsible for the local calculations and transformations. The concept can be considered as a positive result of the unification of different swarm computing approaches. Cloud computing is proposed as an application field for the notion of graph transformational swarms with stationary members. Two illustrative case studies of problems from the field are presented.

Chapter 7 discusses how graph-transformational swarms can be used to model dynamic logistic networks. Motivated by the observation that swarms in nature already solve problems closely related to logistics, the chapter shows how close the main features of dynamic logistic networks and the syntactic and semantic components of graph-transformational swarms are. Thereby, the graphical representation as well as the parallel

semantics of graph-transformational swarms play an important role. To illustrate this, essential aspects of the routing problem of automated guided vehicles are modeled as graph-transformational swarms. The chapter demonstrates the capability of the approach regarding visualization in the design level as well as the computation level.

Hence the contents of this thesis can be summarized in the following table. The table also shows the distribution of the peer reviewed contributions that have been published during the development of this thesis.

Chapter	Title	Remarks
1	Introduction	
2	Swarms and swarm computing	
3	Graph transformation	
4	Graph-transformational swarms	Contains the main part in [3]
5	Unification capability	Contains a part from [3]
6	Stationary members	Contains [4]
7	Modeling of decentralized processes in dynamic logistic networks	Contains [1] and [2]
8	Conclusion	

Chapter 2

Swarms and swarm computing

The motivation behind graph-transformational swarms is not restricted to the unification of existing swarm computing approaches. It should also offer the possibility to develop new kinds of swarm inspired solutions designed for other application areas than the classical use for optimization problems. For this reason, knowledge about biological foundations of swarm computing is also important.

Animals living in groups can solve complex problems based on their cooperation. It is assumed that the complex behavior of swarms emerges from relatively simple rules that are followed by the members (see, e.g., [9, 74, 18, 45, 16, 38]).

This chapter starts by giving an overview of the historical development of the knowledge gained from the observation of some swarm behaviors. It outlines some seminal experiments done in laboratories with real animals as well as the developed models that simulate the real behavior. Such models were a very important milestone in understanding the principles behind swarm behavior as well as in developing swarm computing methods. In the second part, this chapter reviews the major swarm computing approaches focusing on the common ideas and principles. In order to illustrate the diversity of the field, two relatively new methods are also introduced.

This chapter is organized as follows: Section 2.1 provides a compact review on how swarms in nature solve their problems, highlighting two main behaviors: the foraging behavior of ants and the schooling and flocking behaviors as can be observed in fish and birds respectively. The three major computational swarm approaches: cellular automata, particle swarm optimization and ant colony optimization are introduced in Section 2.2. Section 2.3 provides also a brief overview on two other swarm computing approaches namely, artificial immune systems and bees algorithm. Finally, this chapter closes with a summary that includes a discussion about the common ideas of swarms and swarm computing.

2.1 Swarms in nature

The purpose of this section is to give a brief overview on the literature on swarms in nature. It focuses on two main behaviors. The motion behavior which can be found in bird flocks and fish schools as well as the foraging behavior of ants. This overview

highlights some well known studies that are directly related to the computing methods which are introduced in the next section.

2.1.1 Bird flocks and fish schools

When a bird flock moves like a wave turning and gliding around and around, it appears to the observer like a single body that changes its velocity, volume and shape in a cohesive manner (cf. Figure 2.1). Several other group-living animals produce similar collective coordinated motion behavior. Some examples are fish schooling (cf. Figure 2.2), humpback whales hunting and locusts migrating. The common characteristic of these behaviors is the ability to move synchronously and very fast as a group. In biology, this ability is considered to be a result of self-organization which is defined by S. Camazine and colleagues as follows:

... self-organization is a process in which pattern at the global level of a system emerges solely from numerous interactions among the lower-level components of a system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern[9]

Generally, the self-organization theory is well established to explain swarm behavior (i.e., [16, 9, 74]). However, only a little is known yet about the details of the mechanisms responsible for the actual behavior.

For many decades, numerous studies have been carried out to explain the functions and the structures of fish schools and bird flocks ¹. The majority of results in this area has been obtained by observing the behavior of animals in the presence of (simulated) predators or food.

The seminal work of Radakov [85] highlights the role of the informational transfer in a school. Radakov was one of the first scientists that tried to explain school behavior as a result of the propagation of local information through the school and not as a phenomenon coordinated by a leader. He observed that when distributed, the schools of *atherinomorus* exhibit a propagation of information in a form which he coined: *waves of excitation*. These so-called *waves* can be attenuated or amplified by obstacles or interactions in the school. Partridge, Pitcher and colleagues have undertaken a series of studies based on longitudinal observations of the species of European minnows, cod and herring in large circular tanks and flow channels which have produced results similar to those presented by Radakov [74, 73, 81]. They too investigated the communication mechanisms of schools, but focusing this time on the sensorial capacities of their members. They found out that the *lateral line* plays a crucial role in the schooling behavior [73]. The lateral line is a system of organs that sense vibrations in water. Furthermore, they observed that the respective species exhibit swarm behavior without leaders in groups of up to three members. Partridge summarizes these results in his definition of a school:

¹However, studies of fish schooling are more frequent in the literature. This is due to the fact that fish are more easy to observe when maintained in open tanks compared to the observation of birds in flight.



Figure 2.1: A flock of starlings performs an aerobatic display in the south of Scotland before roosting for the night. ²

It is a group of three or more fish in which each member constantly adjusts its speed and direction to match those of the other members of the school.[74]

Patridge has argued that due to water being the medium of movement, the fish in a school are more difficult to detect for predators than when swimming alone. If detected, they can react quickly, building different shapes as a group which confuse the predator [74]. More recent studies have supported the above mentioned observations and thesis but have also complemented them. Swarm behavior can improve the foraging activities in some cases [80]. It also provides hydrodynamic benefits. It has been demonstrated that forming certain school patterns can reduce the energy consumption of the fish [42, 95]. These functional results have been associated mainly with two mechanisms: The many eyes effect of animal aggregation and the rapid propagation of information across the school (cf. [72]).

From observation to modeling As mentioned above, it is accepted in research that the members of a swarm act locally, communicating with neighbors and following rules. However, numerous questions remain hard to answer, such as those pertaining to the exact nature of the rules that the individuals follow and their number, or how to define

²License: CC BY-SA 2.0, Source: <http://www.geograph.org.uk/photo/1069366>, Photographer: Walter Baxter

neighborhood; whether it is determined by the number of the nearest members or by a given radius that the members can sense. It is very difficult in general (in some cases even impossible) to predict the behavior of a system composed of a huge number of subsystems by analyzing the behavior of the subsystem and vice versa. One established method used to answer such questions is mathematical modeling using simulations in computers. One of the best known and first simulations of collective behavior was devised



Figure 2.2: A school of *bigeye trevally* forming a big bait ball.³

by C. Reynolds [87]. He generated the motion of computer-animated flocking *boids*. In his model every individual follows three simple rules which are stated by the author as follows:

Collision avoidance: *avoid collisions with nearby flockmates*

Velocity matching: *attempt to match velocity with nearby flockmates*

Flock centering: *attempt to stay close to nearby flockmates*

The neighborhood is defined locally as a local region. Every boid has a region which is defined by a fixed distance, measured from its center, and a fixed angle, measured from the boid's direction of flight. Boids outside this local neighborhood are ignored (see illustration in Figure 2.3). The resulting animation exhibits realistic-looking flocking and

³License: CC BY-NC-ND 2.0, Source: <https://www.flickr.com/photos/racaza/2450934646/> , Photographer: Raymond

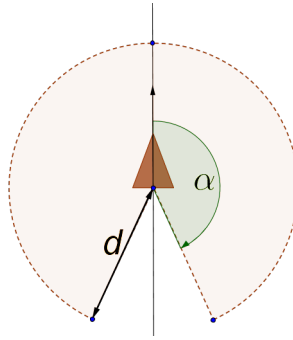


Figure 2.3: The neighborhood of a boid as defined in [Reynolds87]. It is a region specified by a distance d measured from the center of the boid and an angle α measured from the boid's direction of flight

schooling behaviors such as cohesion and polarization. Since the publication of Reynolds animation which was originally developed for entertainment purposes, many other more sophisticated and more rigorous models have been developed (see, e.g., [16, 45, 60]).

2.1.2 Ant colonies

Ants are social insects living in groups. Such groups are called colonies and are highly organized structures composed of units that range from a few hundred to several millions of individuals. In general, the members of an ant-colony are structured in at least three so-called *castes*: *workers*, *drones* and *queens*. The workers make up the majority of the members, and are sterile females. They can also be organized in some specialized subgroups depending on the tasks that they perform [43].

An ant-colony continually solves numerous complex problems by means of division of labour and self-organization. Among others, the members of the colony construct nests, keep them clean, organize reproduction, search for food and transport it to the nest. The method of operation that is discussed in this work deals with the behavior behind this latter activity, also called *foraging behavior*. The foraging behavior has inspired one of the most famous swarm computing approaches namely *ant colony optimization*, which will be introduced in the next section. For computing approaches derived from other activities in ant colonies refer to [7]

Note that foraging behavior differs from one species to another. It also depends on other factors such as environment and food quality (cf., [99]). For some ant species, where the individual's capacities are limited, the indirect communication by means of pheromone trails plays a crucial role. Pheromone is a chemical substance which an ant deposits to mark a path or position in such a way that other ants or the ant itself can retrieve it later. It can, for example, be used as a marker to help ants return to the nest or to attract other foragers to a given food source. Figure 2.4 illustrates the building of a pheromone trail by a colony of *Argentine ants*. Such a cooperation is called *stigmergy*. It describes the indirect communication through applying changes to the environment.

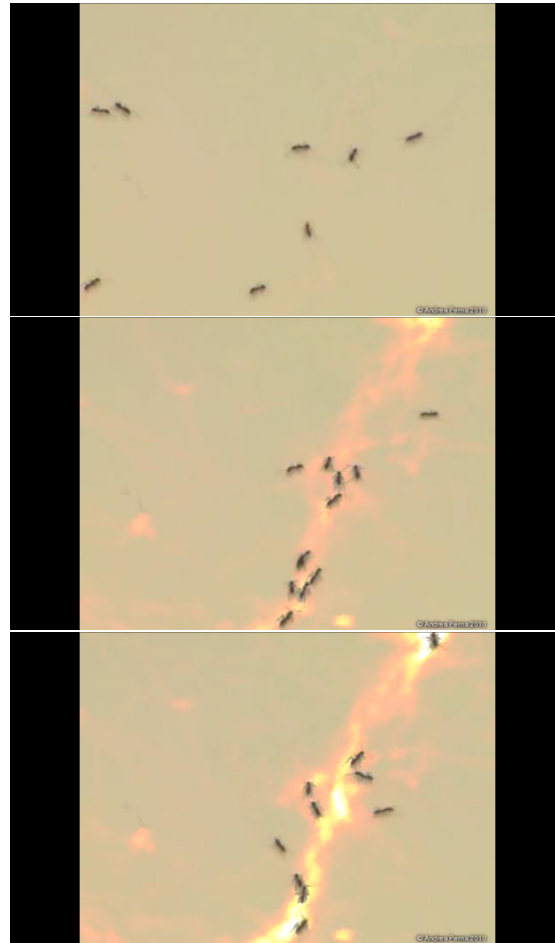


Figure 2.4: The building of a pheromone trail by a colony of Argentine ants. The three images are extracted from a video. The first one is at the beginning of the experiment. The second and the third image are taken respectively at the 0.29 and 1.11 minutes.⁴

To understand the mechanisms behind the foraging behavior and in order to carry out controlled studies many researchers choose to observe ants in laboratories. The work by J.-L. Deneubourg , S. Aron, S. Goss and J. M. Pasteels can be considered as directly related to ant colony optimization. The experiments are carried out in laboratories with colonies of ants from the species *Iridomyrex humilis* known also as Argentine ant (see Figure 2.5).

Compared to other species *I. humilis* ants are very small and possess only a limited individual capacity for orientation. Therefore they are especially well suited for exploring cooperation behavior within groups. Another characteristic of this species is that the ants permanently mark their trail on their way and not only when they have found a food source as is the case with several other species.

In the above mentioned experiments the nest of the colony was linked to the food source

⁴(source:[77]. License: Creative Commons Attribution version 2.5



Figure 2.5: Argentine ant (*Iridomyrex humili*) ⁵

via a bridge with different setups. The main idea was to simplify the bridge in such a way that there are some points where the ants have only a choice between two possible paths: left or right.

Binary bridge experiment Daneubourg et al. used in [18] a simple experimental setup consisting of a bridge with two equally long branches (see Figure 2.6). This experiment is also known as the *binary bridge* experiment. At the beginning of the experiment, the ants choose randomly between the two branches. After an average time of ten minutes one of the branches is selected by most of the ants (about 90% in average). This behavior is explained as a result of the *positive feedback* mechanism: Every ant chooses a branch based on the quantity of pheromone present on it, thereby reinforcing the amount of pheromone on the chosen branch. The selection of one of the branches from the majority of the ants at the end of the experiment is due to the random fluctuations in branch selection.

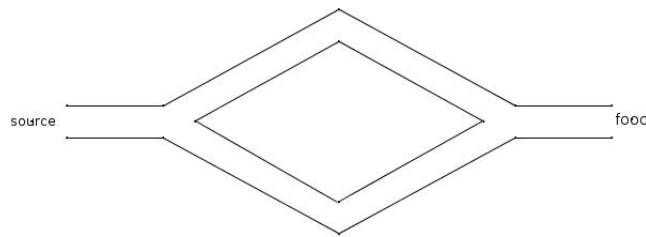


Figure 2.6: *binary bridge* experiment, a sketch of the experiment setup adapted from [18]

The researchers developed a formal model of the observed behavior. In this model and for simplification, it is assumed that every ant deposits the same amount of pheromone called a *pheromone unit*, as well as that the pheromone does not evaporate. The second assumption is quite realistic since the pheromone needs more than half an hour for evaporation and the experiments are completed before this time scales. Let us denote the

⁵source:AntWeb.org .Photographer:April Nobile. License: CC-BY-SA-3.0

branches by B_1 and B_2 , and let $B_i(j)$ be the numbers of ants that have used branches B_i after j ants have traversed the bridge for $i = 1, 2$. Let us define the *complement* function \bar{i} for $i = 1, 2$ given by $\bar{1} = 2$ and $\bar{2} = 1$. The probability that the $(j + 1)$ th ant chooses the Branch $i = 1, 2$ is:

$$P_i(j + 1) = \frac{(k + B_i(j))^\alpha}{(k + B_i(j))^\alpha + (k + B_{\bar{i}}(j))^\alpha} = 1 - P_{\bar{i}}(j + 1) \quad (2.1)$$

where k quantifies the attractiveness of a less visited branch. In other terms, a large k permits more exploration of new knowledge. α determines the importance of the difference in pheromone between the two branches in the decision process. If α is large, even if the branch B_i has a lesser number of pheromone units than $B_{\bar{i}}$, B_i will be chosen by the ant $j + 1$ with high probability. α can therefore be interpreted as a parameter for the exploitation bias of the acquired knowledge. The probability in equation 2.1 is analyzed using a *Monte Carlo* simulation. It was found that the values that give the best fit to the experimental measurements are $\alpha \approx 2$ and $k \approx 20$.

short-path experiment Goss et al. [39] extended the binary bridge experiment, using a bridge composed of two identical modules. Every module consists of a long and a short branch by itself. The two modules are placed one after another as depicted in Figure 2.7. The relation between the lengths of the long and the short branch is defined by a ratio r . It was observed that in a given run, initially, the ants randomly choose between the different possible paths. Over time (approximately 10 min), more and more ants prefer the shortest path. Towards the end of a run, most ants choose the shortest path. Particular, for $r = 2$, more than 80% of the total traffic measured between the 30th and 40th minutes used the shortest path in the bridge. The figure 2.7 gives an illustration of the distribution of ants in this experiment. It was also found that the probability of selecting the shorter path increases with r . These results are explained by the fact that the shortest path between the nest and the food source can be traversed more quickly and frequently. Hence the pheromone density on the shortest path increases in comparison to other paths attracting more and more ants over time. In this case too, the researchers have developed a model that mimics the observed behavior. The probability to choose between two possible branches is based on the equation 2.1 with the fixed parameter values $\alpha = 2$ and $k = 20$.

In summary, the complex behavior of Argentine ant colonies, which consists of finding the shortest path, can be explained by the *positive feedback* mechanism on the individual level. Every ant in the colony lays pheromone (*trail-laying*), and follows paths with high amounts of pheromone (*trail-following*). These results and the model behind them are the basis for the development of the swarm computing method ant systems.

2.2 Major swarm computing methods

So far this chapter has presented an introduction to swarms in nature by means of specific behavior of birds, fish and ants. Now, it is time to introduce the related swarm

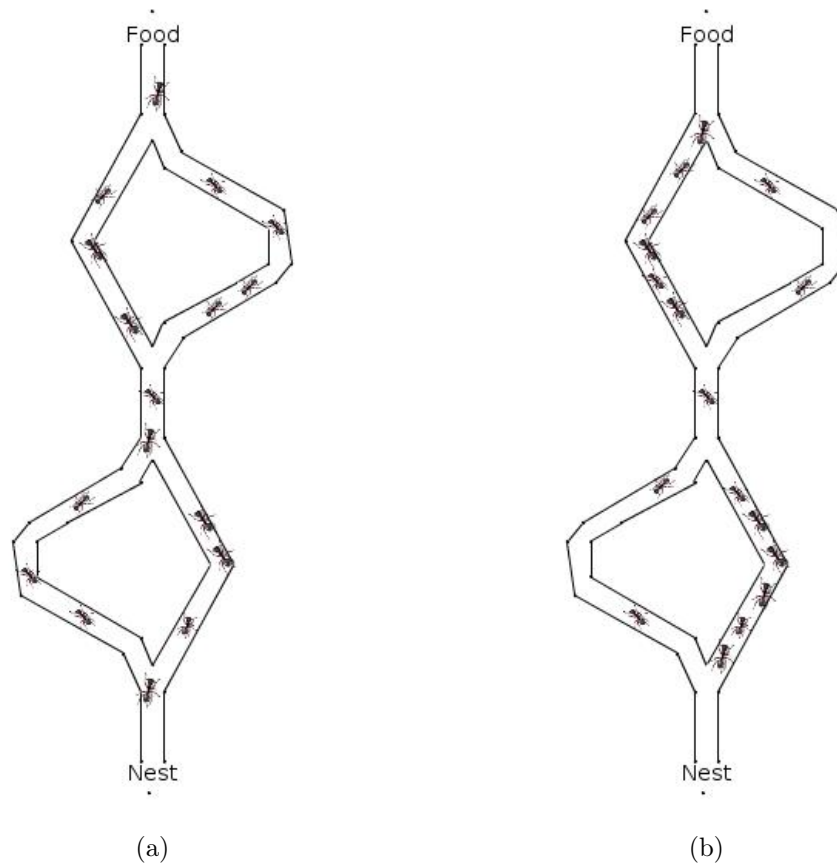


Figure 2.7: The experiment shows that *Argentine ants* select the short branches on both modules of the bridge with a high probability (adapted from [38]) (a) During the first minutes after the placement of the bridge (less than 5 min) the ants seem to choose randomly between short and long parts of the two modules. (b) Later, between 5 and 10 minutes the food is discovered by some ants. In this stage, the traffic on the bridge increases but the choice between short and long parts of the modules is still random. Some minutes later one path of the bridge is clearly preferred.

computing approaches. Namely, particle swarm optimization, ant colony optimization and cellular automata.

2.2.1 Particle swarm optimization

General idea Particle swarm optimization (PSO) is a population-based method motivated from the movement behavior of animal societies like bird flocks and schools of fish. In PSO a number of particles operate in the solution space of an optimization problem. Thus, their positions correspond to the solutions of the problem. Furthermore, every particle can move (change its location) in the given space with a certain velocity (i.e., rate of change). Based on a combination of its own experience and the information from other particles in the swarm, a particle adapts its position and its velocity in every calculation step. Precisely, the new position is changed according to the particles own velocity and the best position found so far (*cognitive component*) combined with the best positions found within its neighborhood (*social component*). Non-determinism also belongs to the method. It is expressed in the form of small perturbations of the personal and social components.

The origins In addition to the swarm behavior of animals described above, the social behavior of humans has also played a major role in the development of particle swarm optimization. It was a motivation of the developers to model this behavior to prove their hypotheses. The authors argue that thinking, mind and intelligence are all concepts that cannot be considered as isolated, private or intern such as the prevailing cognitivist perspectives do. According to the authors, they derive from the social interaction of individuals. People learn from each other through comparing, imitating, sharing experiences and emotions [48, 50].

The main difference between the swarm behavior of animals such as bird flocks and fish schools and the social behavior of humans is the abstractness of the latter. Humans adjust their beliefs and attitudes to conform with other society's members. It can be assumed that the minds "navigate" in a high-dimensional psychosocial space.

The term *particle swarm* results from the authors first experiments with swarms where the resulting visualization shows individuals like particles instead of birds or fish

[...], at some point the two-dimensional plots we used to watch the algorithms perform ceased to look much like bird flocks or fish schools and started looking more like swarms of mosquitoes. The name came as simply as that. [50, page xviii]

The term "particle" is then considered to be suitable because of the notion of position and velocity that was adopted in PSO [50, page xx]. PSO is based on a social-psychological model of social influence and social learning.

Particle swarm optimization as metaheuristic particle swarm optimisation (PSO) was originally designed and proposed by Eberhart and Kennedy in 1995 [49]. The proposed algorithm is considered as the canonical version of PSO and has been used as basis

to generate several other versions (for an overview see for example [82, 105]). In the current section, an overview of the canonical PSO is given. A more formal description can be found later in Chapter 5. The canonical PSO algorithm for a maximization problem can be summarized in Algorithm 1:

Algorithm 1 Canonical particle swarm optimization

- 1: initialize a set of n particles $\{P_i\}_{i \in [n]}$. A particle P_i starts with random position p_i and velocities v_i . It has furthermore a personal best position pb_i which corresponds to the initial position at the beginning.
 - 2: **repeat** for each particle P_i
 - 3: evaluate the current fitness $f(p_i)$ of the current position p_i
 - 4: **if** $f(p_i) > pb_i$ **then**
 - 5: $pb_i \leftarrow p_i$.
 - 6: get the best neighbor bn_i .
 - 7: generate two random vectors rnd_1 and rnd_2 .
 - 8: $v_i \leftarrow v_i + rnd_1 \otimes (pb_i - p_i) + rnd_2 \otimes (bn_i - p_i)$.
 - 9: $p_i \leftarrow p_i + v_i$.
 - 10: **until** terminal condition is met
-

Let us describe Algorithm 1 using the number in front of each line as line reference. Let E be a search space that consists of a d -dimensional euclidean space and let f be a fitness function defined over E . A swarm of n particles acts in E as follows. The position p_i as well as the velocity v_i are initialized randomly for each particle $i = 1, \dots, n$ (Line 1). After that, the following iteration is repeated for each particle i . The fitness function of the current position $f(p_i)$ is calculated and compared with the personal best pb_i (Line 3). If $f(p_i) > pb_i$ then a personal best is found and pb_i is accordingly adapted (Line 4 and Line 5). The best neighbor bn_i is evaluated (Line 6). It consists of the particle with the best fitness value in the neighborhood of i . The form of the neighborhood can be considered as a parameter that should be specified at the beginning of the algorithm. For the non-deterministic behavior of the algorithm two random vectors rnd_1 and rnd_2 are generated (Line 7). They are uniformly distributed in pregiven intervals. The velocity and the position of the given particle are changed according to the equations in Line 8 and Line 9. The symbol \otimes represents the component-wise multiplication. The algorithm ends if a terminal condition is met. Usually, the algorithm terminates if a “sufficiently” good solution is found or a maximum number of iterations has been performed.

Interpretation and geometrical illustration As described in the previous section, the particle swarm optimization has its motivation in modeling social interaction. In this context, the equations introduced above can be interpreted as follows. The equation in Algorithm 1 Line 8 is used to update the velocity v_i , which represents a kind of “memory” of the previous velocity for the particle i . This velocity is changed by adding the *cognitive component* $\Delta 1 = rnd_1 \otimes (pb_i - p_i)$ and the *social component* $\Delta 2 = rnd_2 \otimes (bn_i - p_i)$

leading to the following simplified update formula:

$$v_i^{new} = v_i + \Delta 1 + \Delta 2 \quad (2.2)$$

$$p_i^{new} = p_i + v_i^{new} \quad (2.3)$$

The cognitive component represents the individual memory consisting of the best position found in the past. It quantifies the tendency of the particles to explore their own knowledge. The social component represents the influence of the neighborhood on the individual decision. It quantifies a kind of group standard that individuals try to acquire. The figure 2.8 illustrates the effect of the velocity and position equations for a single particle in a two-dimensional vector space.

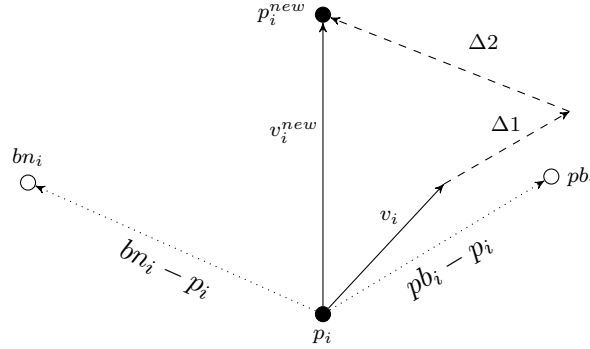


Figure 2.8: Geometrical illustration of the velocity and position update

2.2.2 Ant colony optimization

Ant colony optimization (ACO) is a set of algorithmic concepts based on the ideas gained from the observations of real ants (as described in Section 2.1.2). It was formulated as a metaheuristic to solve discrete optimization problems by Dorigo in [20]. ACO has been proposed as a common framework to cover a wide variety of algorithms inspired by real ant behavior. In fact, since the *Ant System*(*AS*) [21] , which is the first ant based algorithm, several other algorithms that can be considered as an extension of *AS* have been developed and applied successfully to a wide range of combinatorial problems (for overview see [22]). One of the most widespread direct successor of *AS* is the *MAX-MIN Ant System* (*MMAS*). This section recalls the two methods *AS* and *MMAS* using *TSP* as an example application. Before this, a general description of the general ideas of ACO algorithms is given. The section ends up with a more formal presentation of the *ACO* as metaheuristic for solving combinatorial problems.

General ideas of ant colony optimization In ant colony optimization several artificial ants (in the following text only “ants” is used) operate on a graph searching for good solutions of a given optimization problem instance. That is, given a discrete optimization problem one has first to define an adequate representation of its instances as

graphs. Such graphs are called *construction graphs* and contain pheromone values and other problem-specific information. The ants construct solutions incrementally by walking on the construction graph. The movement from one node to another is stochastic following pheromone and heuristic values in the neighboring nodes and edges. This decision is based on an exploration/exploitation process. Meaning that the ants choose the next node in every step in proportion to the pheromone value of the corresponding edge (exploitation of the knowledge of the whole system). It is permitted however, that the ants choose nodes with lower pheromone amount. The pheromone values are changed during the runs. The main idea of the algorithm is that the edges belonging to good solutions have the tendency to receive more visits from ants and carry higher pheromone values than other edges. This corresponds to the *positive feedback* mechanism that the real ants follow (see Section 2.1.2). Two questions are behind the development of different versions of ACO. What does the strategy that the ants follow to choose the next nodes look like exactly? And how are the pheromone values updated?

Ant system *Ant System* (*AS*) is the first version of the ACO algorithms. It was proposed by Dorigo et al. [21]. It was initially designed to solve the *traveling salesperson problem* (*TSP*). In TSP, an instance corresponds to a pair $(G, dist)$, where G is a complete⁶ directed graph and $dist : E_G \rightarrow \mathbb{R}^+$ assigns to every edge a weight called *distance*. A feasible solution corresponds to a *Hamiltonian cycle*. And the goal is to minimize the cost function of a Hamiltonian cycle which corresponds to the sum of distances of the edges belonging to the cycle.

The algorithm adapts the ant behavior to solve the TSP problem as follows: At the beginning every edge of the *construction graph* is initiated with the same (virtual) pheromone value τ_0 . A number of ants n are randomly assigned to starting nodes. Each ant a builds its own tour by repeatedly choosing the next node to be visited. At a given current node i , the probability p_e^a to choose a possible next edge e is given by the equation:

$$p_e^a = \frac{\tau_e^\alpha \cdot \eta_e^\beta}{\sum_{f \in J_i^a} \tau_f^\alpha \cdot \eta_f^\beta} \quad (2.4)$$

where τ_e is the pheromone value on the edge e and η_e is a heuristic value equal to the inverse of the distance $dist(e)$ (i.e. $\eta_e = \frac{1}{dist(e)}$). J_i^a denotes the set of possible edges for the ant a starting from i , which is equal to the edges outgoing from i and not yet visited by ant a . The parameters α and β control the relative influence of the pheromone respectively the heuristic value.

After that every ant a constructs a Hamiltonian cycle H^a with the cost $cost^a$, the pheromone quantity of an edge e is updated using the following equation:

$$\tau(e) \leftarrow (1 - \rho) \cdot \tau(e) + \sum_{a=1}^n \Delta \rho_e^a \quad (2.5)$$

⁶not complete graphs can be transformed to fully connected graphs, by connecting the unconnected nodes with edges having very large distances

where $\Delta\rho_e^a$ is the value of pheromone that ant a deposits on the edge e . It is defined as follows:

$$\Delta\rho_e^a = \begin{cases} 1/\text{cost}^a & \text{if edge } e \in H^a \\ 0 & \text{otherwise} \end{cases} \quad (2.6)$$

In other words, only the pheromone values of the edges visited by the ants a are increased with an amount equal to the inverse of the cost cost^a of the constructed cycle H^a . The idea behind the equation 5.7 is that the edges belonging to Hamiltonian cycles with lower costs receive more pheromone. By means of the probability equation in 5.5, these edges are more likely to be chosen by ants in the next iterations. *AS* has been tested on small TSP instances (up to 70 nodes). The results have revealed that the algorithm can solve instances with up to 30 cities with acceptable performance, compared with other heuristics. However, it has shown its limit for larger instances [21]. This was another factor for the development of other *ACO* algorithms with better performances.

MAX – MIN Ant System *MAX – MIN Ant System (MMAS)* introduces modifications to the *Ant System* in the way how the update of pheromone is performed. The solution construction using the equation 5.5 as described above is kept the same. First, only the edges belonging to the best (or to the best so far) cycle are enhanced. To prevent rapid stagnation on sub-optimal solutions, a second modification is introduced by *MMAS*. It consists of the definition of an interval $[\tau_{min}, \tau_{max}]$ to limit the range of pheromone values.

According to this, the update equation in *MMAS* for a given edge e looks as follows:

$$\tau(e) \leftarrow [(1 - \rho) \cdot \tau(e) + \Delta\rho(e)]_{\tau_{min}}^{\tau_{max}} \quad (2.7)$$

where $\Delta\rho(e) = 1/\text{cost}_b$ if e belongs to the best *walk* or 0 else. ρ is a parameter called the evaporation rate. The notation $[]_{\tau_{min}}^{\tau_{max}}$ means that the value between the brackets will be replaced by τ_{max} if it exceeds τ_{max} and respectively if the value is lower than τ_{min} then it will be replaced by τ_{min} . In addition *MMAS* have proposed many other technical improvements like the initialization of the pheromone by τ_{max} or the reinitialization of the pheromone values if the algorithm stagnates.

2.2.3 Cellular automata

Historically, *cellular automata* are one of the first computing approaches that exploit the knowledge and metaphors of biology. The research on cellular automata dates back to the late 1940s in the studies by von Neumann. The concept of cellular automata is inspired by inter cellular communication in biological systems exploring the idea that complex computations can be performed by simple cells acting in parallel. Cellular automata have been established in computer science for many decades as computational devices with massive parallelism (see, e.g., [12, 47, 101, 102, 104]). They are also considered as typical representatives of swarm computation (cf. [50]).

A cellular automaton is a system composed of elementary entities called cells which are organized as a network. Each cell exhibits a state which is a value in a countable set. The

value of a cell is adapted based on some local rules that use the states of the neighboring cells to change the current state. Usually, the local rules are the same for each cell and do not change over time. The state of the whole automaton is called a configuration. A current configuration can change into a follow-up configuration by the simultaneous changes of all local states using the cells local rules. To keep the technicalities simple, we consider two and one-dimensional cellular automata. In both cases, a cell is represented as a square in the Euclidean space.

In the theory of cellular automata, the most commonly used neighborhoods are the *von Neumann* and the *Moore* neighborhood. Both neighborhoods are characterized by a range r that specifies how far the neighboring cells are away from each other. To illustrate this, let us consider two dimensional cellular automata which are represented by a grid. Let us imagine that there is a kind of agent ⁷ that can move from one cell to the next one in every step. Starting from a cell c , the set of cells that can be reached in r steps constitutes the neighborhood. In the von Neumann neighborhood the agent is restricted to move only in vertical and horizontal directions. In the Moore neighborhood, the agent has no restrictions. Figure 2.9 displays the von Neumann (a) and Moore (b) neighborhood respectively for $r = 1, 2, 3$. In both examples, the cells marked by r belong to the neighborhood of c which has a range r for $r = 1, 2, 3$. In the case where $r = 1$ the corresponding cells are additionally drawn with a thick line.

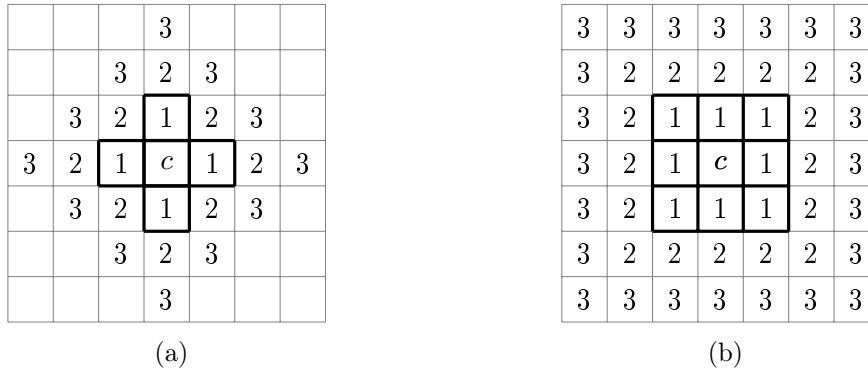


Figure 2.9: Two-dimensional (a) von Neumann and (b) Moore neighbors of cell c in the ranges 1, 2 and 3.

One of the most popular examples of a cellular automaton is the *Game of Life* [36]. Game of life is a two-dimensional cellular automaton with two states *black* (which stands for *alive*) or *white* (standing for *dead*). A cell changes its state using the following four rules:

- if alive and there are less than two alive neighbors, then die (loneliness);
- if alive and there are more than three neighbors, then die (overcrowding);
- if dead and there are three alive neighbors, then comes to life (reproduction);

⁷The most common example in the case of the von Neumann neighborhood is a taxi. The underlying distance is also called Manhattan distance and refers to the distance of path that a taxi takes between any two places in Manhattan.

- otherwise, stay unchanged.

With these simple rules one can generate complex patterns starting from initial ones. In the literature one can find different kinds of initial patterns that produce interesting behavior during the computation process. There are some patterns that oscillate, blink, grow or move across the grid. Figure 2.10 displays an example of a pattern called *glider* that moves in 18 computational steps from the bottom left corner in the north-eastern direction.

Beside this fascinating visual effect that emerges from simple rules, the game of life can be used to perform computations to solve a given problem. It was also demonstrated that game of life using some special initial pattern can emulate a universal Turing machine (see for example the study in [86]).

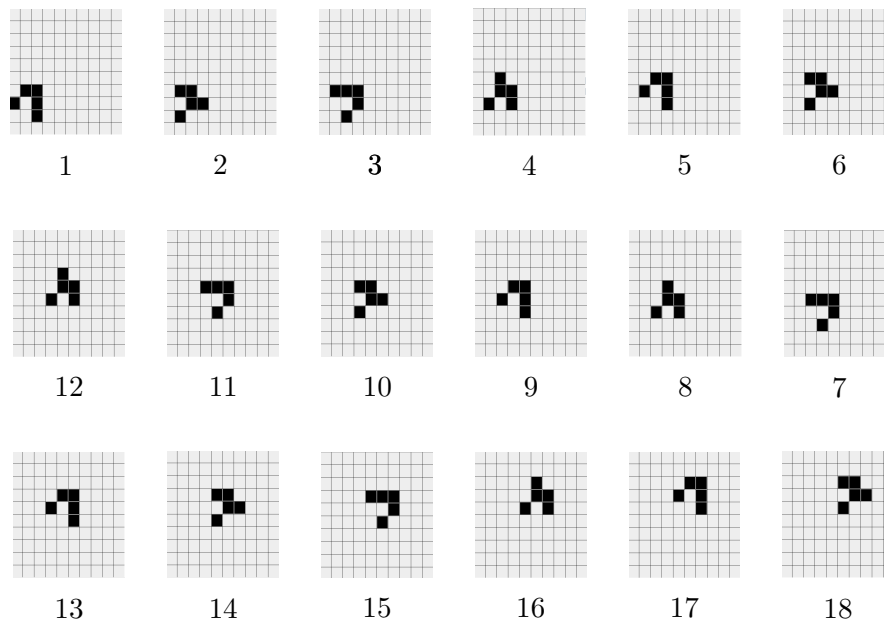
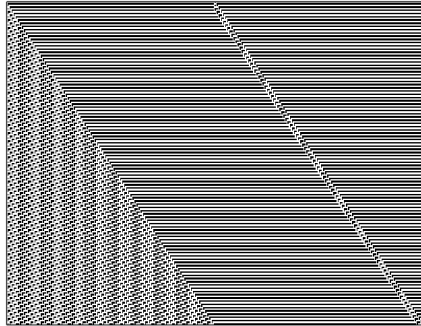


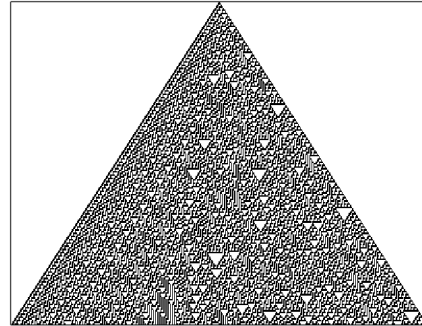
Figure 2.10: Example of a *glider* that moves in north-east direction

Another famous example of cellular automata is the elementary cellular automaton. It is a one-dimensional cellular automaton, the cells of which have two states, black or white. Each cell changes its state based only on its current state and the state of immediate neighbors (the left and right cells). There are 256 different possible rules which are well studied in the literature. Figure 2.11 displays the history of four automata of which uses each a different rule, namely the rules 9, 30, 110 and 150. Each row of pixels represents a generation in the history starting from the top $t = 0$ which is initialized by a black cell in the middle of the row. Given an arrow t , the next arrow $t + 1$ represents the result of the application of the underlying rule from all cells in arrow t until the bottom of the grid is reached (in this case 200 steps are needed). Like game of life, Elementary cellular automata also exhibit a universal propriety. Actually, it has been proven that

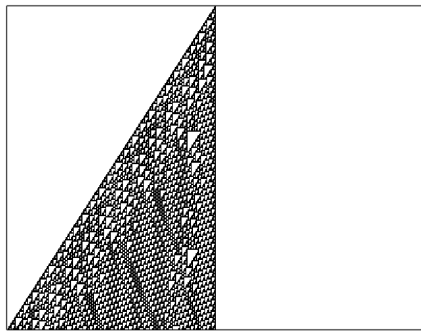
the automaton using the rule 110 is capable of universal computation [13]. For more details on elementary cellular automaton refer to [104].



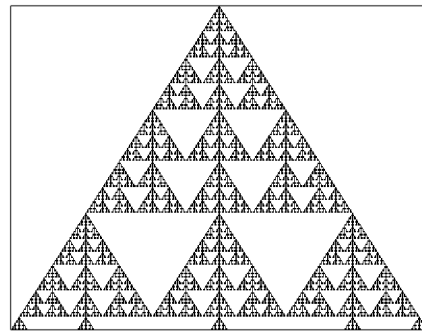
(a) Rule 9



(b) Rule 30



(c) Rule 110



(d) Rule 150

Figure 2.11: Four examples of the history of elementary cellular automata.

2.3 Other methods

2.3.1 Artificial immune systems

Artificial immune systems (AIS) are swarm computing methods inspired by the principles and metaphor of the vertebrate immune system. The immune system of vertebrate is composed of cells and molecules that collaborate with each other and with other systems to maintain the host body operational. It protects the body by detecting a wide variety of infectious agents, known as *pathogens*, such as viruses, bacteria, viroid and other parasites. Pathogens exhibit specific *antigens* on their surfaces that provoke reactions from the immune system. The way in which the immune system responds in such a detection can have mainly two forms. The *innate* or *adaptive* immunity. The innate immunity plays an initiatory and preparatory role. It reacts in a generic way

immediately against any pathogens. The adaptive immunity complements the innate one by generating adapted responses to specific pathogens. It consists mainly of white blood cells known as *lymphocytes*, more specifically *B* and *T*-lymphocytes.

One of the major methods that characterize the field of artificial immune systems is the *clonal selection*. (For an overview of the other methods in the field of artificial immune systems see e.g., [10]).

Clonal selection designs a class of algorithms inspired by the clonal selection theory [8]. The main idea behind this theory is that a specific antigen activates only a unique specific *B*-lymphocyte. When an activation occurs, the organism clones the activated cell by means of cell division creating similar cells. In this way, it produces a high quantity of the specific antibodies. During the cloning process the resulting antibodies increase their affinity for the underlying antigen. This phenomenon is known as *affinity maturation*. It is caused by a mutation and a selection mechanism. The division of cells leads to the production of two kinds of cells. The *plasma cells* and the *memory cells*. The process of maturation in both kinds is called *differentiation*. While *plasma cells* consist of terminal (non-dividing) antibody secreting cells and play a central role in the immediate response of the active antigens, the memory cells are reserved for an ulterior antigenic stimulus. They are able to generate high affinity antibodies, pre-selected for the specific antigen responsible for the primary activation. Figure 2.12 illustrates the clonal selection principles.

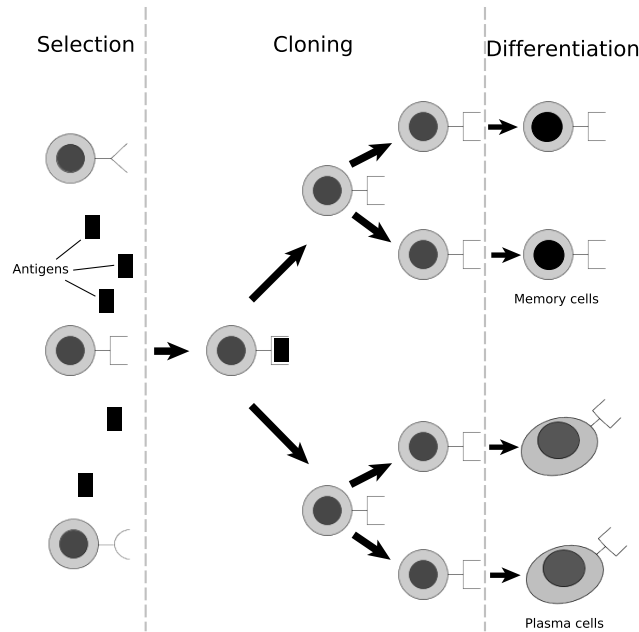


Figure 2.12: Clonal selection principle adapted from [10]

Any approach that uses computational techniques inspired by the affinity maturation process of B-cells can be considered as a clonal selection inspired approach. One of the

most popular ones is the CLONALG [17] algorithm. CLONALG has been proposed to solve machine-learning and pattern recognition tasks. It has also been adapted to solve optimization problems.

Given a set S of patterns to be recognized, CLONALG considers the elements in S to be antigens. The goal is to produce a set of memory antibodies M , that match the members in S . After creating a random set of antibodies A , the algorithm repeats two main procedures: (1) Generating f clones of elements in A which mimics the proliferation of B-cells creating a number of clones for an antibody proportional to its affinity and (2) mutation, which mutates attributes of the created antibodies. The antibodies in A with the highest affinity are added into the memory set M . A number n (which is a parameter to be fixed at the beginning) of antibodies with the lowest affinity in A are replaced with new randomly generated antibodies.

2.3.2 Bees algorithm

The *bees algorithm* is a swarm computing method inspired by the food foraging behavior of honey bee colonies [79]. A bee colony starts foraging searching randomly for promising fields. The promising fields correspond for example to flower patches with high amounts of nectar or pollen. The foragers -called *scout bees* in the underlying algorithm- which found a promising field perform the so called “waggle dance” in a specified place called “dance floor”. A waggle dance encodes three indications regarding a flower patch. The distance from the hive, the direction to follow and the quality.

The bees algorithm integrates these ideas to solve optimization problems in the following way. At the beginning of the search process a small number n of scouts randomly explores the solution space for solutions with high fitness. Within a *recruitment* procedure a simulation of the waggle dance of biological bees is used to communicate findings to the other foragers. The procedure assigns to every solution a number of foragers in proportion to its fitness. The foragers then perform a *local search* in the neighboring region of the assigned solution. If the local search for a given region stagnates, the region with its local fitness optimum is abandoned. In the algorithm a small number of scouts keep exploring the solution space looking for new regions of high fitness (global search), Repeating the procedures above for the new regions.

The algorithm was designed for both combinatorial optimization and functional optimization. However until now it has been mainly used for the latter. The obtained results are very promising regarding the robustness against trapping in local optima.

2.4 Summary

This chapter has provided an overview of the collective behavior of animals and swarm computing approaches inspired thereby. How ant colonies forage for food differs obviously from bird flocking or fish schooling. It was shown, however, that the mechanisms behind the two behaviors seem to be similar⁸. In both behaviors, the individuals are self-

⁸Despite of the mentioned similarity of underlying mechanisms, the diversity of biological systems should be stressed. Animals living in groups and exhibiting swarm behavior are not an exception. Even

organized. They follow simple rules (compared to the resulting behavior) and they communicate using local information collected from their neighbors or from the environment. In a specific experimental setup, the behavior of a given species has been successfully simulated. Besides confirming the self-organization assumption, these simulations have opened a door to the development of swarm computing approaches offering the basic mathematical models. In this chapter the three major swarm computing approaches ant colony optimization, particle swarm optimization and cellular optimization were introduced. It was attempted to determine the general ideas as well as the main components that characterize every approach. Furthermore, the two relatively new swarm computing approaches artificial immune systems and bees algorithm were briefly introduced.

In an attempt to propose a unifying approach, it can be stated in this summary that the presented approaches exhibit the following general ideas: A swarm consists of several members that act in an environment in parallel (at least theoretically). Therein, the environment is modeled in such a way as to permit an easy coding of the communication between the members as well as between the members and the environment. Each member follows simple rules by updating its state using update equations. These equations can contain non-deterministic factors to simulate the stochastic behavior of biological systems. In contrast to swarms in nature, swarm computing approaches have some centralized control⁹, that regulate the behavior of the whole algorithm toward reaching a goal. In Chapter 4 a more systematic analysis of the common ideas and components of swarms in nature and swarm computing is given as well as a formulation using methods of graph-transformation. For that an introduction to a suitable graph transformational approach is needed which follows in the next chapter.

if one consider ants by themselves, the foraging behavior differs from species to species. For example, *Sahara Desert ants* forage for food using totally different navigation skills than Argentine ants. A Sahara Desert ant navigates using memorized angles in respect to the Sun. Furthermore, it can count its steps in grounds where pheromone quickly vanishes using a sort of internal pedometer.

⁹with the exception of CA

Chapter 3

Graph transformation

This chapter gives an overview on graph transformation focusing on the basic elements as far as needed in this thesis. We consider directed edge-labeled graphs and their derivation by application of rules. The graph transformation approach is chosen in such a way that rules can be applied in parallel and that their parallel applicability follows from the applicability of each of the involved rules and additional conditions. Moreover, the notion of graph transformation units which comprise a set of rules and a control condition is used. Such a unit is a computational device that models the derivation of graphs while the control condition is obeyed. Units are used in the next chapters as swarm's members and parallelism makes sure that the members can act simultaneously. The present introduction is based on a set-theoretical approach.

3.1 General ideas and backgrounds

Graphs are very common and well suited means to represent connected structures. Graph transformation offers a formalism that extends the statical description of graphs to include the dynamic changes based on rule applications.

The research in the area of graph transformation has been initiated in the late sixties and early seventies with applications in rule-based image recognition and processing as well as graph generation and translation (see e.g., [78, 84]). One of the main motivations therein has been the need for a generalization of Chomsky grammars over graphs [15]. As in Chomsky grammars the rule-based modification is the basic idea in graph transformation paradigms. To illustrate what a rule based-modification over graphs look like, let us consider the following informal description. Given a graph G , and a rule r specified by two graphs $r = (L, R)$, the application r to G consists of finding L in G and replacing it by R , leading to a graph H (see Figure 3.1). This illustration presents the main idea of a rule-based derivation over graphs, however at the same time, it shows that, unlike strings in Chomsky grammars, making replacements in graphs is not obvious. Indeed, some important details are needed in order to introduce a formal definition are missing. For instance, what do the underlying graphs precisely looks like? How to find a given graph L in G ? How to disconnect L from G ? How to connect R in order to construct H ?

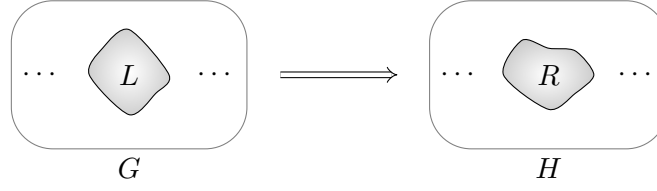


Figure 3.1: Schematic presentation of a graph transformation

In the literature, there are several manners to handle those questions. This implicates the existence of different graph transformation approaches (see for example [24, 54, 59]. For an overview see [88]). This work considers directed edge-labeled graphs with derivations based on the *double-pushout* approach. Which is one of the first and most used approaches in graph transformation field. This approach is introduced based on the set theory using similar notations as in [54]. Even though this chapter has a recalling character, it also provides a very important theorem concerning parallel rule applications, namely the generalized parallelization theorem.

This Chapter is organized as follows: Section 3.3 introduces the kind of graphs used along this thesis as well as some operations that permit their manipulation. In Section 3.4 rules and their applications are presented. Section 3.5 extends the notion of applicability proposing the concept of application context. In Section 3.6 parallelism for rules and their application is studied. Section 3.7 recalls first shortly the definition of a *graph grammar* as a first structuring of rules and their applications. It introduces then the concepts of graph class expressions and control conditions, which are used in the definition of graph transformation units. Subsequently, different graph transformational tools are introduced and compared in Section 3.8. The chapter ends with a summary in Section 3.10.

3.2 Preliminaries

The reader is assumed to be familiar with the usual basic notions and notations of set theory. But the disjoint union is explicitly introduced as it is a less conventional operation that is heavily used in this thesis.

Definition 1

Let $F = (X_i)_{i \in I}$ be a family of sets such that I is a countable set. A set X is a disjoint union of F with respect to the injective mappings $(in_i : X_i \rightarrow X)_{i \in I}$ if:

- (i) $in_j(X_j) \cap in_k(X_k) = \emptyset$ for $j \neq k$ and
- (ii) $\bigcup_{i \in I} in_i(X_i) = X$

General assumption 1

The disjoint union of a family $F = (X_i)_{i \in I}$ of sets exists and is denoted by $\sum_{i \in I} X_i$. For two sets A and B , the disjoint union is denoted by $A + B$.

Note that there is no unique construction of the disjoint union but it is characterized up to bijections in the following way.

Remark 1

1. Let $b : X \rightarrow Y$ be a bijection. The set Y is also a disjoint union of F with respect to the mappings $(in'_i : X_i \rightarrow Y)_{i \in I}$ with $in'_i = b \circ in_i$ for $i \in I$.
2. If a set Y is a disjoint union of F with respect to the mappings $(in'_i : X_i \rightarrow Y)_{i \in I}$ then X and Y are bijective (i.e., there is a bijective mapping $b : X \rightarrow Y$ defined as $b(x) = in'_i(x)$ for $x \in in_i(X_i)$).

Moreover, two important properties of disjoint unions are needed. Both concern the disjoint union of subsets. Figure 3.2 gives a schematic illustration of the components used there.

Remark 2

1. Let $F_X = (X_i)_{i \in I}$ and $F_Y = (Y_i)_{i \in I}$ such that $Y_i \subseteq X_i$ for all $i \in I$ and X is disjoint union of F_X with respect to $(in_i^X : X_i \rightarrow X)_{i \in I}$. Then $Y = \bigcup_{i \in I} in_i^Y(Y_i)$ is disjoint union of F_Y with respect to $(in_i^Y : Y_i \rightarrow Y)_{i \in I}$ defined by $in_i^Y(x) = in_i^X(x)$ for $x \in Y_i$. By construction, we have $Y \subseteq X$.
2. Let $F_X = (X_i)_{i \in I}$ and $F_Y = (Y_i)_{i \in I}$ such that $Y_i \subseteq X_i$ for all $i \in I$. Let Y be a disjoint union of F_Y with respect to $(in_i^Y : Y_i \rightarrow Y)_{i \in I}$. Then there is a disjoint union X of F_X with respect to the mappings $(in_i^X : X_i \rightarrow X)_{i \in I}$ with $in_i^X(x) = in_i^Y(x)$ for $x \in Y_i$. By construction, we have $Y \subseteq X$.

3.3 Graphs

All graphs considered in this thesis are directed and have labels on edges.

Definition 2 (graphs)

Let Σ be a set of labels. A (*directed edge-labeled*) *graph* over Σ is a system $G = (V, E, s, t, l)$ where V is a set of *nodes*, E is a set of *edges*, $s, t : E \rightarrow V$ and $l : E \rightarrow \Sigma$ are mappings assigning a *source* $s(e)$, a *target* $t(e)$ and a *label* $l(e)$ to every edge $e \in E$.

Σ may contain the symbol $*$ which stands for *unlabeled* and is omitted in drawings of graphs. An edge e with $s(e) = t(e)$ is a *loop*. If $e \in E$ is labeled with z , e is also called a *z-edge* or a *z-loop* respectively. An edge with label $*$ represents an *unlabeled edge*. The components V , E , s , t , and l of G are also denoted by V_G , E_G , s_G , t_G , and l_G , respectively. The empty graph is denoted by \emptyset . The class of all directed edge-labeled graphs is denoted by \mathcal{G}_Σ . Given a graph $G \in \mathcal{G}_\Sigma$, the notation $x \in G$ denotes that x is a node or an edge of G , i.e., $x \in V_G$ or $x \in E_G$.

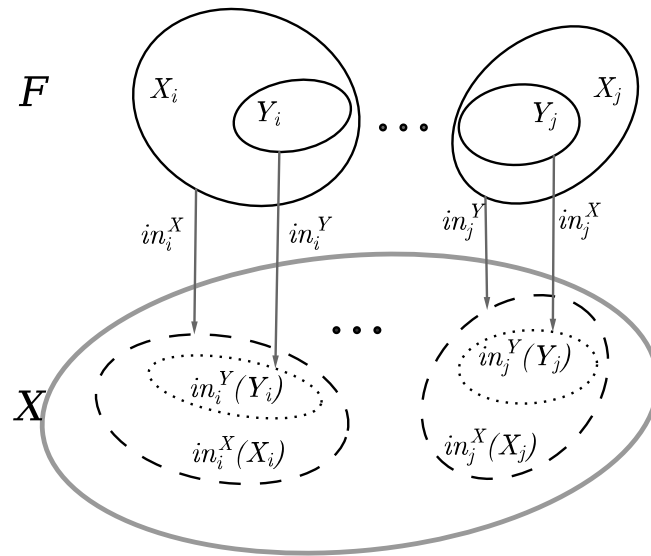


Figure 3.2: An illustration of disjoint union construction

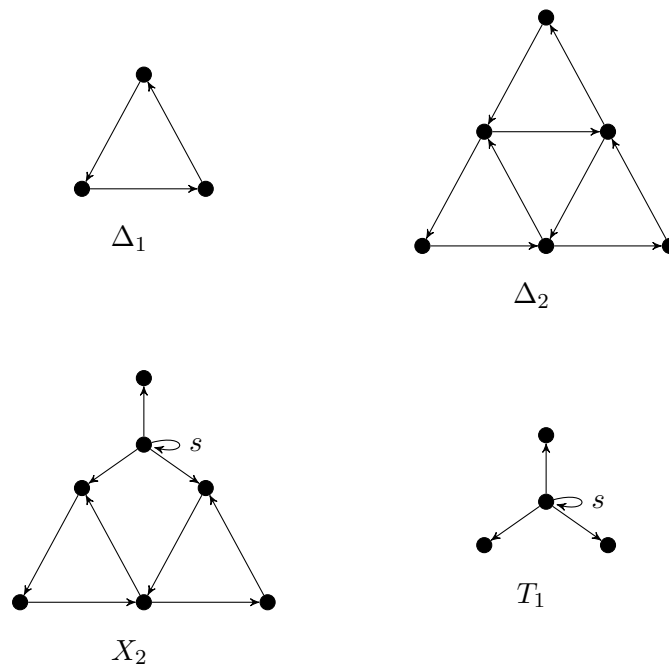


Figure 3.3: Examples of graphs

Example 1 (graphs)

Figure 4.1 displays four simple graphs that are used below for different illustrations.

At this point, some operations and relations that permit performing changes on graphs can be introduced. The first relation is *subgraph* which is defined as follows.

Definition 3 (subgraph)

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

Example 2

Consider the graphs T_1 and X_2 in Figure 4.1. T_1 is a subgraph of X_2 provided that the underlying nodes and edges are named accordingly. Actually, there are six ways to do so because the tripod in X_2 allows six permutations.

In order to introduce rule applications in the chosen framework, three additional operations are needed. The first operator is the *subtraction*. It permits the removal of nodes and edges from a given graph. Note that the resulting structure of a subtraction is not necessarily a graph, since the removal of nodes can cause dangling edges. To overcome this, the so called *contact condition* is introduced. It guarantees that the subtraction produces a subgraph of the underlying graph. The second operation is the *extension* of graphs by a specific structure. The third operation is the *disjoint union* which permits the construction of graphs using other graphs.

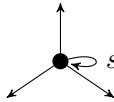
Definition 4 (subtraction)

Given a graph $G \in \mathcal{G}_\Sigma$ and a pair of sets of nodes and edges $X = (V_X, E_X)$, such that $(V_X, E_X) \subseteq (V_G, E_G)$. The *subtraction* of X from G is given by $G - X = (V_G - V_X, E_G - E_X, s, t, l)$ with $s(e) = s_G(e)$, $t(e) = t_G(e)$ and $l(e) = l_G(e)$ for all $e \in E_G - E_X$.

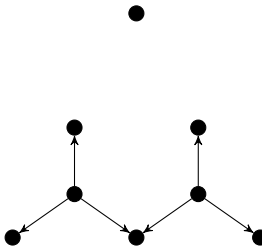
Given a graph G and a subgraph H . Then $G - H$ denotes $G - (V_H, E_H)$.

Example 3

Let Ext be the set of target nodes in T_1 . Then

$$T_1^- = T_1 - (Ext, \emptyset) =$$


and

$$X_2 - T_1^- =$$


In order to ensure that the resulting structure of a subtraction is a graph, an additional condition is needed. Namely the *contact condition*. It guarantees that there are no dangling edges after the removal of nodes.

Definition 5 (contact condition)

Given a graph $G \in \mathcal{G}_\Sigma$ and a pair of sets of nodes and edges $X = (V_X, E_X)$ such that $(V_X, E_X) \subseteq (V_G, E_G)$. The subtraction $G - X = (V_G - V_X, E_G - E_X, s, t, l)$ satisfies the *contact condition* if there is no edge $e \in G - X$ with $s(e) \in X$ or $t(e) \in X$.

If the contact condition is satisfied, the result of subtraction is a subgraph of the original graph.

Example 4

In the example above, the subtraction $T_1 - (Ext, \emptyset)$ does not satisfy the contact condition because the resulting structure T_1^- has three edges without targets. In contrary, the subtraction $X_2 - T_1^-$ satisfies the contact condition because there is no resulting dangling edges.

Definition 6 (extension)

Given a graph $G \in \mathcal{G}_\Sigma$ and a structure $X = (V_X, E_X, s_X, t_X, l_X)$ where V_X is a set of nodes, E_X a set of edges and $s_X: E_X \rightarrow V_G + V_X$, $t_X: E_X \rightarrow V_G + V_X$, $l_X: E_X \rightarrow \Sigma$ are three mappings. The extension of G by X is given by the graph $G + X = (V_G + V_X, E_G + E_X, s, t, l)$ with $f(e) = f_G(e)$ if $e \in E_G$ and $f(e) = f_X(e)$ otherwise for $f \in \{s, t, l\}$.

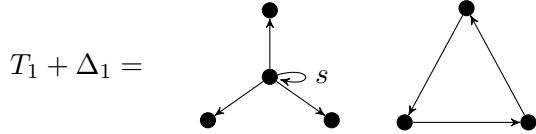
An important operation similar to extension, but between two graphs is the disjoint union of graphs.

Definition 7 (disjoint union)

The *disjoint union* of two graphs G and H is defined as $G + H = (V_G + V_H, E_G + E_H, s, t, l)$ and, for $f \in \{s, t, l\}$, $f(e) = f_G(e)$ if $e \in E_G$ and $f(e) = f_H(e)$ otherwise.

Example 5

The disjoint union of T_1 and Δ_1 can be displayed as:



Another important ingredient is the notion of a graph morphism.

Definition 8 (graph morphism)

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

If g_V and g_K are bijective, g is an *isomorphism* and G and H are called *isomorphic*.

For a graph morphism $g: G \rightarrow H$, the image of G in H is called a *match* of G in H , i.e., the match of G with respect to the morphism g is the subgraph $g(G) \subseteq H$. If the mappings g_V and g_E are injective, the match $g(G)$ is also called *injective*. In this case, G and $g(G)$ are isomorphic.

For $G' \subseteq G$, the morphism $g|_{G'}: G' \rightarrow H$ denotes the *restriction* of g to G' .

Example 6

Let us consider the graphs Δ_1 and Δ_2 as introduced above. There are twelve possible graph morphism from Δ_1 to Δ_2 that are all injective. The first four graph morphism map the triangle Δ_1 to each of the sub-triangles of Δ_2 . The rest is obtained by making additionally rotations around the center of the underlying triangles. Among the graph morphisms of T_1 to itself, there are also non-injective ones. Besides the six permutation that define isomorphisms, one can also map the three edges that are not loops on two or one of them in various ways.

3.4 Rules and their applications

As mentioned in the introduction two main components are needed in order to define a rule: A component L which has to be found in the graph where we want to apply the rule, and a second component R which replaces L . In the double pushout approach a third component K is needed to ensure an adequate linking to the surrounding graph. In addition L , K and R are all graphs and K is a subgraph of L and R .

Definition 9 (rule)

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

In the following, the class of all rules is denoted by \mathcal{R} .

Example 7

The Figure 3.4 shows two examples of rules. The first one is called *delete* and has a gluing graph that consists of two nodes: A *start* node represented by an unfilled square and an *end* node represented by an unfilled circle.

Intuitively, it can be argued that the rule deletes (as the name indicates) an occurrence of the *middle* node represented by a filled circle in L and their attached edges and replaces them by an edge from the start node to the end node. How this is performed exactly is explained below.

The second example consists of the rule called *terminate*. In this rule the gluing graph consists of the target nodes of the left-hand side. The right-hand side consists of these nodes connected by edges to form a triangle as Δ_1 . This rule is used below to generate interesting forms.

The first step in the application of a rule (L, K, R) to a graph G is to find a valid match $g : L \rightarrow G$ of the left-side graph in G . The validity is determined by two conditions that g should satisfy. The first condition is the *contact condition*. It ensures that the removal of the part $g(L) - g(K)$, which occurs in the rule application as introduced in Definition 11, yields a subgraph of G . Precisely it guarantees that there will be no dangling edges after the removal of nodes in $g(V_L) - g(V_K)$. The second condition is the *identification condition*. Considering that g can be noninjective, it is possible that two different items in L have the same image using g . The identification condition requires that such items should be elements in the image of the gluing graph K . This condition

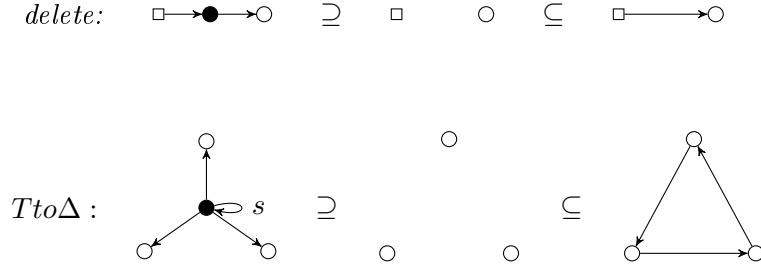


Figure 3.4: graph transformation rules

is necessary for the Parallelization Theorem below. Both conditions together are called the *gluing condition*.

Definition 10 (gluing condition)

Let $r = (L, K, R)$ be a rule and $g : L \rightarrow G$ a graph morphism. g satisfies the *gluing condition* if the following two conditions are satisfied.

1. The *contact condition*: g satisfies the contact condition if the subtraction $G - (g(L) - g(K))$ satisfies the contact condition as introduced in Definition 5, i.e., there is no edge $e \in E_G - (g(E_L) - g(E_K))$ with $s(e) \in (g(V_L) - g(V_K))$ or $t(e) \in (g(V_L) - g(V_K))$.
2. The *identification condition*: If two nodes or edges are identified via g they must belong to the gluing graph K , i.e., $g(x) = g(x')$ for $x, x' \in L$ implies $x = x'$ or $x, x' \in K$.

Remark 3

The contact condition can also be formulated as follows: if there is a node $v \in g(L)$ which corresponds to a source or a target of an edge $e \in E_G - (g(E_L) - g(E_K))$ then $v \in g(V_K)$.

Example 8

Figure 3.5 shows two examples where the contact and identification conditions are not satisfied. Both examples use the same rule $r = (L, K, R)$ and the match of L is surrounded by a gray ellipse. The match of L in the graph G_1 via the morphism g_1 does not satisfy the contact condition because the edge labeled by e has as target a node that is not in the subtraction $G_1 - (g_1(L) - g_1(K))$. The match of L in the graph G_2 using g_2 violates the identification condition because the middle and right nodes in L are identified by g_2 , however the middle node does not belong to the gluing graph K .

Now we have all ingredients to define a rule application.

Definition 11 (rule application)

The application of a rule $r = (L, K, R)$ to a graph $G = (V, E, s, t, l)$ consists of the following three steps.

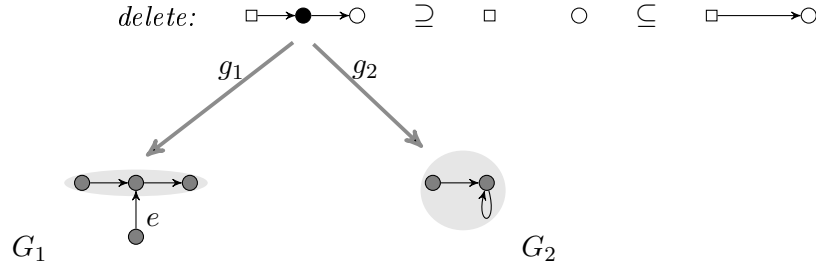


Figure 3.5: Matches that violate the gluing conditions.

1. Choose a match $g(L)$ of L in G subject to the *contact condition* and the *identification condition*.
2. Remove the match of L up to $g(K)$ from G , using the subtraction operation. The result is an intermediate graph $Z = G - (g(L) - g(K))$.
3. Add the right-hand side R to Z by gluing Z with R in $g(K)$ yielding the graph $H = Z + X_{R-K}$ with $X_{R-K} = (V_{R-K}, E_{R-K}, s_X, t_X, l_R)$ and for $e \in E_{R-K}$ and $f \in \{s, t\}$ $f_X(e) = g(f_K)$ if $f_R(e) \in V_K$ and $f_X(e) = f_R$ otherwise. In other words, all edges keep their labels. For each edge $e \in E_R - E_K$ which loses its source or target due to the operation $V_R - V_K$, it is redirected to the image of its original source or target. All other edges keep their sources and targets.

The Figure 3.6 gives a graphical interpretation of an application of a rule (L, K, R) to a graph G using a morphism g . The first step corresponds to find a match $g(L)$ in G represented there as a circle which is composed itself of two parts. A ring (filled with horizontal lines) which stands for $g(K)$ and an inner circle (filled with vertical lines) standing for $g(L) - g(K)$. The representation of $g(K)$ as a ring reflects the role of the gluing graph and the contact condition. Namely, they guarantee an adequate linking to the surrounding graph which is represented by dots in the figure. The deletion of $g(L) - g(K)$ in step 2 yields to a graph Z with a white inner circle. The graph H represents the resulting graph after the application of the rule. The inner circle (filled with oblique lines) corresponds to the added part $(R - K, g)$ as defined in step 3 of the rule application. The graph morphisms d and h are given by $h(v) = d(v) = g(v)$ for $v \in V_K$, $h(e) = d(e) = g(e)$ for $e \in E_K$, $h(v) = v$ for $v \in V_R - V_K$, $h(e) = e$ if $e \in E_R - E_K$.

Example 9

Figure 3.7 shows two examples of rule applications. We use the same names of nodes as in Example 7. In the first step of the rule application, a match $g(L)$ that satisfies both the contact and the identification conditions is chosen. In the second step the image of the elements in $g(L) - g(K)$ are deleted yielding to a temporary graph Z . In the third step the elements in $R - K$ are added with respect to the construction described above.

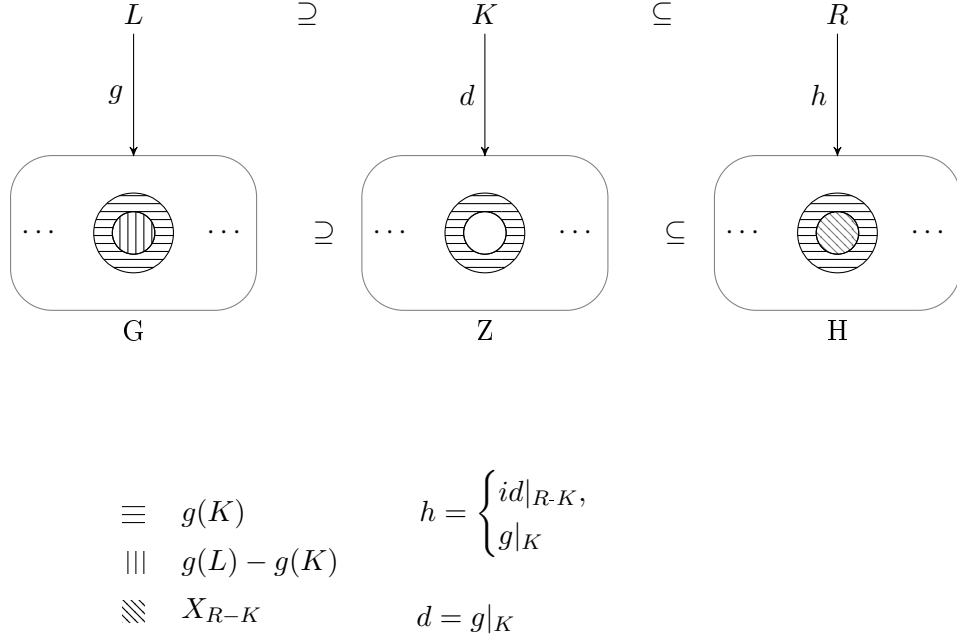


Figure 3.6: A graphical interpretation of application of a rule $r = (L, K, R)$ to a graph G yielding to a graph H

A rule $r = (L, K, R)$ can also be depicted alternatively to the classical presentation $L \supseteq K \subseteq R$ as $L \rightarrow R$. However, it is necessary in such a presentation to deduce unambiguously the graph K . To identify K , generally the same relative positions of nodes in L and R are used, and if necessary different shapes and colors are also used. The rule *delete* can be depicted alternatively as in Figure 3.8.

The application of r to G with respect to the graph morphism g is denoted by $G \xRightarrow[r]{g} H$. It is called a *direct derivation* from G to H . The subscript r may be omitted if it is clear from the context. Sometimes, the morphism g is included in the notation as $G \xRightarrow[r, g]{g} H$. The sequential composition of direct derivations $G = G_0 \xRightarrow[r_1]{g_1} G_1 \xRightarrow[r_2]{g_2} \cdots \xRightarrow[r_n]{g_n} G_n = H$ ($n \in \mathbb{N}$) is called a *derivation* from G to H . As usual, the derivation from G to H can also be denoted by $G \xRightarrow[n]{P} H$ where $\{r_1, \dots, r_n\} \subseteq P$, or just by $G \xRightarrow[*]{P} H$. The string $r_1 \cdots r_n$ is the *application sequence* of the derivation.

Example 10

Figure 3.9 shows two examples of derivations. The derivation with application sequence *delete delete* and the derivation with application sequence *terminate*.

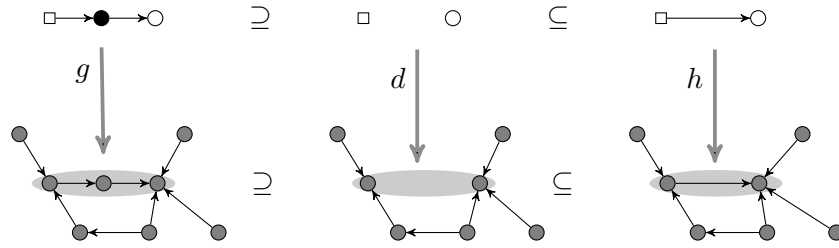
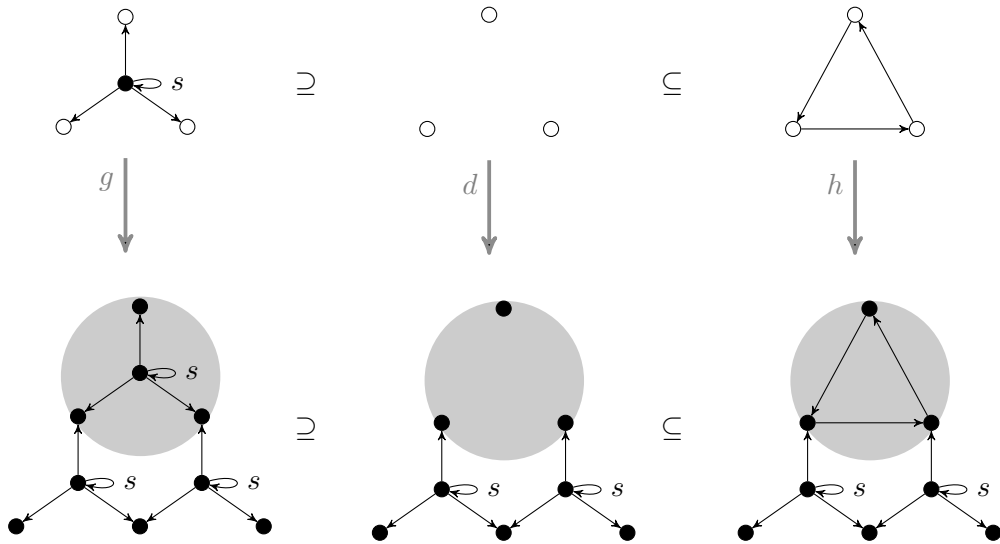
(a) An application of the rule *delete*(b) An application of the rule *terminate*:

Figure 3.7: Application of rules

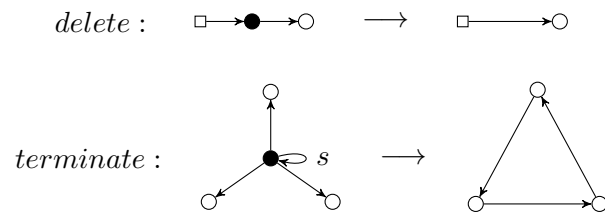
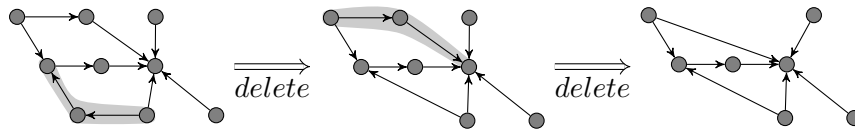
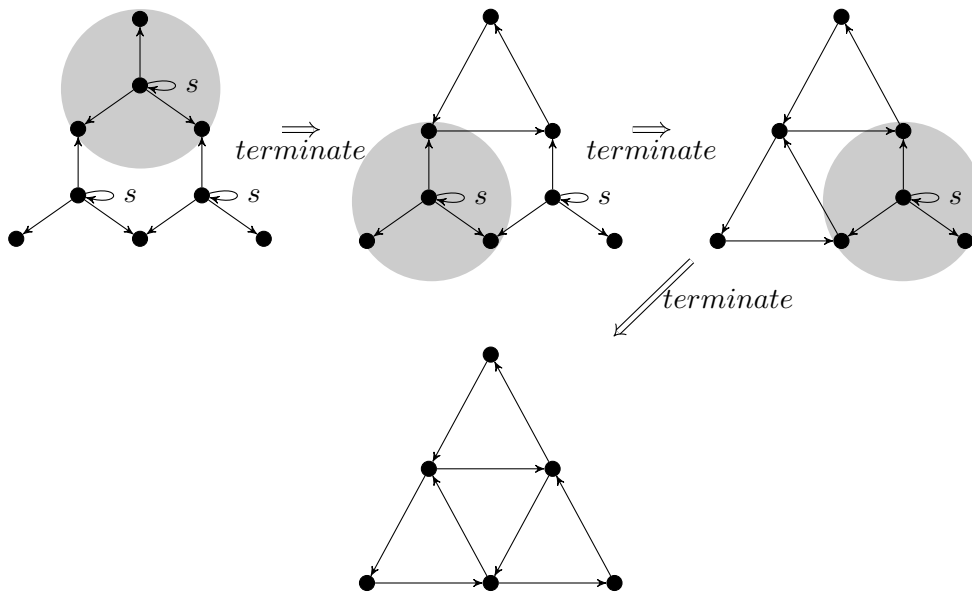
Figure 3.8: Alternative depiction of the rules *delete* and *terminate*(a) A derivation using two successive application of the rule *delete*(b) A derivation using three successive application of the rule *terminate*:

Figure 3.9: Derivations examples

3.5 Application context

Graph transformation using the rules as introduced in previous subsections is computationally complete (For its completeness cf. [54]). However it is sometimes useful to have additional application conditions in order to restrict the applicability of a given rule. In this section two kinds of application conditions are considered, positive and negative contexts. While a positive context can be represented by a unique graph that specifies what should be present in the host graph G in order to apply the given rule, the negative context is composed of a finite number of graphs each specifying a negative constraint. i.e. what should be not present in G . Considering the possibility to represent the application contexts graphically, the approach presented in this section is related to the negative applications as introduced in [40].

Definition 12 (application context)

An *application context* $C = (C_P, C_N)$ over a rule $r = (L, K, R) \in \mathcal{R}$ consists of two components, a graph $C_P \in \mathcal{G}_\Sigma$ called the *positive context* and the *negative context* $C_N = \{N_1, \dots, N_k\}$ which is a set composed of a finite number $k \in \mathbb{N}$ of graphs ($N_i \in \mathcal{G}_\Sigma$ for $i \in [k]$), each one called a negative constraint. Thus, the left hand side graph should consist of a proper subset of the positive context if this one is not empty as well as all negative constraints. i.e., $L \subset C_P$ if $C_P \neq \emptyset$ and for $L \subset N_i$ for $i \in [k]$.

If not empty the positive context specifies a positive part $C_P - L$ which consists of the items of C_P that do not belong to L . Analogously, every negative constraint specifies a negative part $N_i - L$ which consists of the items of N_i that do not belong to L .

An application context over a rule restricts the derivation conditions in the following way

Definition 13 (derivation with application context)

Given an application context $C = (C_P, C_N) = (C_P, \{N_1, \dots, N_k\})$ defined over a rule $r = (L, K, R) \in \mathcal{R}$, a graph $G \in \mathcal{G}_\Sigma$ and a graph morphism $g : L \rightarrow G$. The production $G \xRightarrow[r, g]{C} H$ satisfies the application context C if

1. the morphism g can be extended to the graph C_P and
2. the morphism g cannot be extended to any of the graphs N_i with $i \in [k]$.

The satisfaction of an application context C can be included in the notation of a derivation as follows: $G \xRightarrow[r, g]{C} H$

Note that the application of a rule $r = (L, K, R)$ without an application context is semantically equivalent to its application considering the application context (L, \emptyset) .

Depiction conventions For visual purposes, it is useful to integrate the application context of a rule in its graphical presentation. Consider a rule $r = (L, K, R)$ and a corresponding application context $C = (C_P, \{N_1, \dots, N_k\})$. The rule and its application

context can be depicted as $C \longrightarrow R$ where C is represented as a graph with subgraph L and extra information such that the negative contexts and the positive condition are identified. In this representation, the dashed items belong to the negative part. The positive part is depicted in bold with a gray color different from L and R . The remainder is L . If the negative part contains more than a single edge, then it will be enclosed by a dotted line. In this way, all negative constraints and the positive context are easily recognized. Let us illustrate how to depict rules together with their application contexts. Let us start by recalling the rule in Figure 3.11 which was already introduced in previous section as reference.

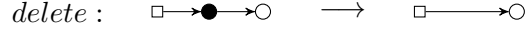


Figure 3.10: A rule without application context

Figure 3.11 shows an example of a rule with negative parts $delete_1$. The negative context consists of a unique constraint $C_N = \{N_1\}$. In the illustration the negative part $N_1 - L$ is dashed. L can be identified easily and unambiguously, since L consists of the part that is not dashed in the left graph.

This negative context requires that the start node has no incoming edge. In other words, the rule can be applied only if it exists a match such that the image of the start node has no incoming edges.

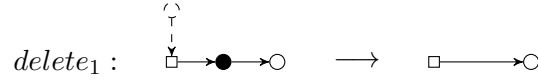


Figure 3.11: A rule with a unique negative constraint where its negative part has one edge

In Figure 3.11 we have a unique negative constraint which specifies a negative part having one edge. In the case where a negative part has more than one edge, it is encircled by a dashed line. For example in Figure 3.12 the negative context requires that the start node or the end node have no incoming edge, meaning that if one constraint is broken, the rule can not be applied.

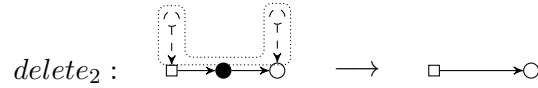


Figure 3.12: A rule with a negative constraint where its negative part has two edges

Figure 3.14 shows an example of a rule with a positive context. The positive part is depicted in bold with another gray color as L . The condition requires that the rule can be applied only if the first node and the end node have outgoing edges.

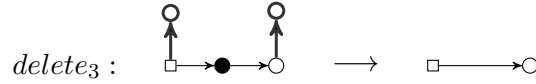


Figure 3.13: A rule with positive constraint

The last example as depicted in Figure 3.14 shows a rule with a positive and a negative context. The negative context is composed of two negative constraints, each of which has one edge. The positive context is similar to the example above.

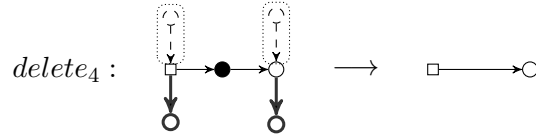


Figure 3.14: A rule with positive and negative contexts

3.6 Parallel rule application

This section introduces parallelism in the chosen graph transformational approach. It is shown under which conditions two or more direct graph transformation derivations can be applied in parallel.

Given two or more rules, it is easy to build a rule through the disjoint union of their components. The resulting rule is called a *parallel* rule.

Definition 14 (parallel rule)

Let $(r_i = (L_i, K_i, R_i))_{i \in I}$ be a family of rules for a countable set I of indices. Then a *parallel rule* $p = \sum_{i \in I} r_i = (L, K, R) = (\sum_{i \in I} L_i, \sum_{i \in I} K_i, \sum_{i \in I} R_i)$ is given by the disjoint unions of the components with injective mappings $in_i^L: L_i \rightarrow L$, $in_i^K: K_i \rightarrow K$ and $in_i^R: R_i \rightarrow R$ for $i \in I$ such that $in_i^K = in_i^L|_{K_i} = in_i^R|_{K_i}$.

The fact that $K \subseteq L$ and $K \subseteq R$ is, without loss of generality, a direct consequence of Remark 2. Actually, one can first construct the disjoint union K of the components K_i . Based on Remark 2 (2.), the disjoint unions L and R exist such that $K \subseteq L$ and $K \subseteq R$. Another way to construct the parallel rule is to start with the disjoint union L and then construct K based on Remark 2 (1.) with $K \subseteq L$ and then use Remark 2 (2.) to build R based on K with $K \subseteq R$.

Example 11

Figure 3.15 shows the parallel rule $delete + delete$. The parallel rule is a composition of the same rule $delete$. The injective mappings are chosen accordingly, so that the resulting components are disjoint. To illustrate the disjunction, different shapes are used.

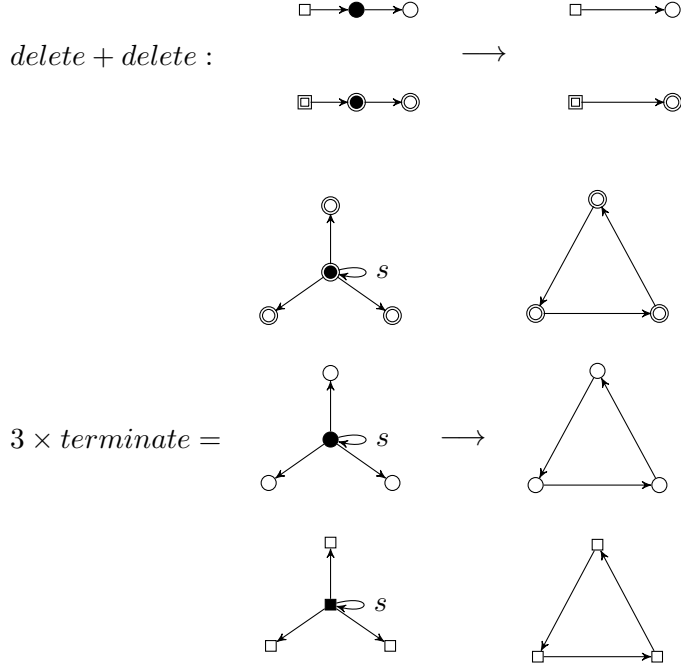


Figure 3.15: Two examples of parallel rules

Definition 15 (parallel independence)

Let $r = (L, K, R)$ and $r' = (L', K', R')$ be two rules and let $G \xRightarrow[r]{\quad} H$ and $G \xRightarrow[r']{\quad} H'$ be two direct derivations with respect to the morphisms $g: L \rightarrow G$ and $g': L' \rightarrow G$. Then the direct derivations are *parallel independent* if the corresponding matches intersect in gluing items only. i.e., $g(L) \cap g'(L') \subseteq g(K) \cap g'(K')$.

Now, we have all ingredients to introduce the generalized parallelization theorem.

Theorem 1 (Generalized Parallelization Theorem)

Let $(r_i = (L_i, K_i, R_i))_{i \in I}$ be a family of rules for a countable set I of indices and let $p = (L, K, R) = (\sum_{i \in I} L_i, \sum_{i \in I} K_i, \sum_{i \in I} R_i)$ be a corresponding parallel rule. Then the following holds true.

1. Let $G \xRightarrow[p]{\quad} X$ be a direct derivation. Then there are direct derivations $G \xRightarrow[r_i]{\quad} H_i$ that are pairwise parallel independent.
2. Let $G \xRightarrow[r_i]{\quad} H_i$ for $i \in I$ be direct derivations. Let each two of them be parallel independent. Then there is a direct derivation $G \xRightarrow[p]{\quad} X$.

Proof.

Let us assume that the parallel rule is constructed using the following injective mappings

$in_i^L: L_i \rightarrow L$, $in_i^K: K_i \rightarrow K$ and $in_i^R: R_i \rightarrow R$ for $i \in I$. Let g be the graph morphism used in the derivation $G \xRightarrow[p]{\quad} X$ and g_i the graph morphisms used in the derivations $G \xRightarrow[r_i]{\quad} H_i$ such that $g_i = g \circ in_i^L$ for $i \in I$

1. For the first direction, we show that for $i \in I$ (a) the matches $g_i(L_i)$ exist in G ; (b) g_i satisfy the identification and (c) the contact conditions; (d) The corresponding derivations are pairwise parallel independent.

(a) **The existence of $g_i(L_i)$** The graph morphism $g_i: L_i \rightarrow G$ is defined by $g_i = g \circ in_i^L$. Hence, the match $g_i(L_i)$ is well defined as $g_i(L_i) = g(in_i^L(L_i))$.

(b) **The identification condition satisfaction of g_i :** Let $j \in I$. To show the assertion, we suppose that $g_j(x) = g_j(x')$ holds for $x, x' \in L_j$ and show that either $x = x'$ or $x, x' \in K_j$ holds true.

Using the definition of g_j , $g_j(x) = g_j(x')$ implies $g(in_j^L(x)) = g(in_j^L(x'))$. Hence, the identification condition of g requires that $in_j^L(x) = in_j^L(x')$ or $in_j^L(x), in_j^L(x') \in K$. Because in_j^L is injective, $in_j^L(x) = in_j^L(x')$ implies $x = x'$ showing the first assertion. The second case is equivalent to $in_j^L(x) \in K$. Hence, there exists $y \in K_k$ with $k \in I$ such that $in_j^L(x) = in_k^K(y)$ and $g_j(in_k^K(y))$ is well defined due to $K \subset L$. By the definition of the parallel rule we have $in_k^K = in_k^L|_{K_k}$, which implies $in_j^L(x) = in_k^L(y)$. The disjoint union construction requires that $in_j^L(L_j) \cap in_k^L(L_k) = \emptyset$ if $k \neq j$ implying that $j = k$. That is, $in_j^L(x) = in_j^L(y)$ for all $x, y \in K_j$. Using again the injectivity of in_j , we get $x = y$ which implies that $x \in K_j$. Analogously, if we assume $in_j^L(x) \in K$, there exists $y \in K_k$ with $k \in I$ such that $in_j^L(x) = in_k^K(y)$ and we obtain $x' \in K_j$ by an identical string of arguments showing the second assertion.

(c) **The contact condition satisfaction of g_i :** Let $j \in I$ and consider $v \in g_j(L_j)$ such that there is an edge $e \in G - (g_j(L_j) - g_j(K_j))$ for $K_j \subset L_j$ with $v = f(e)$ where $f = s_G$ or $f = t_G$. Then we have to show that $v \in g_j(K_j)$ holds true.

Considering $e \in G - (g_j(L_j) - g_j(K_j))$ we obtain that either $e \in G - (g_j(L_j))$ or $e \in g_j(K_j)$ holds true. Using the definition of g_j , this reveals the two cases $e \in G - g(in_j^L(L_j))$ or $e \in g(in_j^K(K_j))$.

In the first case, $e \in g(in_j^K(K_j))$ implies that $e \in g(K)$. Since g satisfies the contact condition, we obtain $v \in g(K)$.

In the second case, $e \in G - g(in_j^L(L_j))$ implies that $e \in G - g(L)$ or $e \in g(in_k^L(L_k))$ such that $k \neq j$, $k, j \in I$. Considering $e \in G - g(L)$, we can use that g satisfies the contact condition to obtain $v \in g(K)$. If $e \in g(in_k^L(L_k))$ such that $k \neq j$, then the structure preservation property of g reveals $v \in g(in_k^L(L_k))$. Hence, we have $v \in g(in_k^L(L_k)) \cap g(in_j^L(L_j))$. Now, by the disjoint union construction we have $in_k^L(L_k) \cap in_j^L(L_j) = \emptyset$. Therefore, the contact condition satisfaction of g requires that $v \in g(K)$.

In all considered cases we have $v \in g(K)$ which implies that there is $f \in I$ such that $v \in g(in_f^K(K_f))$. Using the definition of parallel rule; which requires that $in_f^K = in_f^L|_{K_f}$, we get $v \in g(in_f^K(K_f)) = g(in_f^L(K_f))$. Utilizing the assumption $v \in g(in_j^L(L_j))$ together with the disjoint union construction of L , we obtain $j = f$. Hence, we have that $v \in g(in_j^L(K_j)) = g_j(K_j)$, i.e., g_j satisfies the contact condition.

- (d) **Pairwise parallel independence of g_i :** Consider two different derivations $G \xRightarrow[r_j]{r_k} H_j$ and $G \xRightarrow[r_k]{r_j} H_k$ using the morphisms g_j and g_k , for $j, k \in I$, $k \neq j$ respectively. To prove pairwise parallel independence, we show that $g_j(L_j) \cap g_k(L_k) \subseteq g_j(K_j) \cap g_k(K_k)$.

Suppose $y \in g_j(L_j) \cap g_k(L_k)$. Hence, there are $x_j \in in_j^L(L_j)$ and $x_k \in in_k^L(L_k)$ such that $y = g(x_j) = g(x_k)$. The disjoint union construction requires that $in_j^L(L_j) \cap in_k^L(L_k) = \emptyset$. Hence, the identification condition of g implies that $x_j, x_k \in K$. Together, we have $x_i \in K \cap in_i^L(L_i)$ for $i \in \{j, k\}$.

Let us demonstrate that $K \cap in_i^L(L_i) = in_i^K(K_i)$ for $i \in I$.

Let us start with the inclusion $K \cap in_i^L(L_i) \supseteq in_i^K(K_i)$. The definition of parallel rule requires that $in_i^K(K_i) = in_i^L(K_i)$. Hence, $K_i \subseteq L_i$ implies $in_i^K(K_i) \subseteq in_i^L(L_i)$. Since by definition of K $in_i^K(K_i) \subseteq K$, $K \cap in_i^L(L_i) \supseteq in_i^K(K_i)$ holds true.

The second inclusion can be shown using a proof by contradiction. Let us assume that there is an item x such that $x \in K \cap in_i^L(L_i)$ and $x \notin in_i^K(K_i)$. $x \in K$ and $x \notin in_i^K(K_i)$ implies, based on the disjoint construction of K , that there is $f \in I$ with $x \in in_f^K(K_f)$ such that $f \neq i$. According to the definition of parallel rule, $x \in in_f^L(K_f)$. Hence $x \in in_f^L(K_f)$, $x \in in_i^L(K_i)$ and $i \neq f$ which is in contradiction to the disjoint union construction of L which requires that $in_f^L(K_f) \cap in_i^L(K_i) = \emptyset$ if $i \neq f$. Thus, our assumption is false i.e. $K \cap in_i^L(L_i) \subseteq in_i^K(K_i)$.

All together, we get $y \in g(in_j^K(K_j) \cap g(in_k^K(K_k)))$ implying that $y \in g_j(K_j) \cap g_k(K_k)$, which shows the assertion.

2. For the second direction we have to show that the match $g(L)$ in G exists and the gluing conditions of the morphism g are satisfied.

- (a) **Existence of the match $g(L)$ in G :** To define the function g , we introduce the mapping $\overline{in_i^L} : \text{Im}(in_i^L) \rightarrow L_i$. Since the injective map in_i^L is also surjective on its image $\text{Im}(in_i^L)$, the map $\overline{in_i^L}$ represents the inverse of in_i^L on this subset. Hence, we can define g via $g(x) := g|_{in_i^L}(x) = g_i \circ \overline{in_i^L}(x)$ for $x \in L_i$ and all $i \in I$.
- (b) **The contact condition of g :** Consider a node $v \in g(L)$ such that there exists an edge $e \in G - (g(L) - g(K))$ and $v = f(e)$ where $f = s_G$ or $f = t_G$. Then we have to show that $v \in g(K)$ holds true.

Given $e \in G - (g(L) - g(K))$, we have that either $e \in G - (g(L))$ or $e \in g(K)$ holds true. Let us consider the two cases separately.

In the first case, $e \in g(K) = g(\sum_{i \in I})$ implies that there exists $f \in I$ such that $e \in g(in_f^K(K_f)) = g_f(K_f)$. Utilizing the structure preservation characteristics of g_f , we can conclude that $v \in g_f(K_f)$ holds. This implies that there exists $x_f \in K_f$ such that $v = g_f(x_f) = g(in_f^K(x_f))$ holds. By construction of the parallel rule we have $in_f^K(x_f) \in K$, which in turn allows us to conclude $v \in g(K)$.

In the second case, $e \in G - (g(L))$ implies $e \in G - (g_i(L_i))$ for all $i \in I$. Now consider $j \in I$ to be arbitrary but fixed. Using the contact condition of g_j we get $v \in g_j(K_i)$, i.e., $v \in g(in_j^K(K_i) \subset g(K)$. This implies that $v \in g(K)$.

We have shown that all cases imply $v \in g(K)$ and hence that g satisfies the contact condition.

- (c) **The identification condition of g :** Consider two elements $x, x' \in L$ such that $g(x) = g(x')$. The construction of g implies that there exist $a \in L_i$ and $a' \in L_j$ such that $g(in_i^L(a)) = g(in_j^L(a'))$, i.e. $g_i(a) = g_j(a')$ with $j, L \in I$. Here, we distinguish two cases.

If $i = j$, then the identification condition of the derivation using the morphism g_i implies that the $a = a'$ or $a, a' \in K_i$. Together with the injectivity of in_i^K and the construction of K , the latter fact implies that $x = x'$ or $x, x' \in K$.

If $i \neq j$, then the parallel independence of the derivations using g_i and g_j implies that $g_i(a) \in g_i(K_i)$ and $g_j(a') \in g_j(K_j)$, which allows us to conclude that $x, x' \in K$.

□

Remark 4

In the graph transformation field, the parallelization theorem is one of the most important results. It has been stated by Kreowski in [52] as a fundamental characteristic of the double push-out approach. It has then been used as a basis for many research studies in the field with adaptation to different approaches.

The generalized parallelization theorem proposes a generalization to the parallelization theorem considering a set of direct derivations rather than only two direct derivations.

Example 12

Figure 3.16 illustrates parallel independent direct derivations and their parallelization. In the first example (above) the two direct derivations, each using the rule *delete*, are parallel independent since their matches are disjoint. According to the parallelization theorem, the rules can be applied in parallel using the parallel rule *delete + delete*. Conversely, the existence of a derivation using the parallel rule *delete + delete* implies that each component rule *delete* can be applied separately.

Similarly, the second example illustrates the parallelization using derivations based on the rule *terminate*. The three direct derivations, each using the rule *terminate*, are also parallel independent since their matches intersect only on gluing graphs.

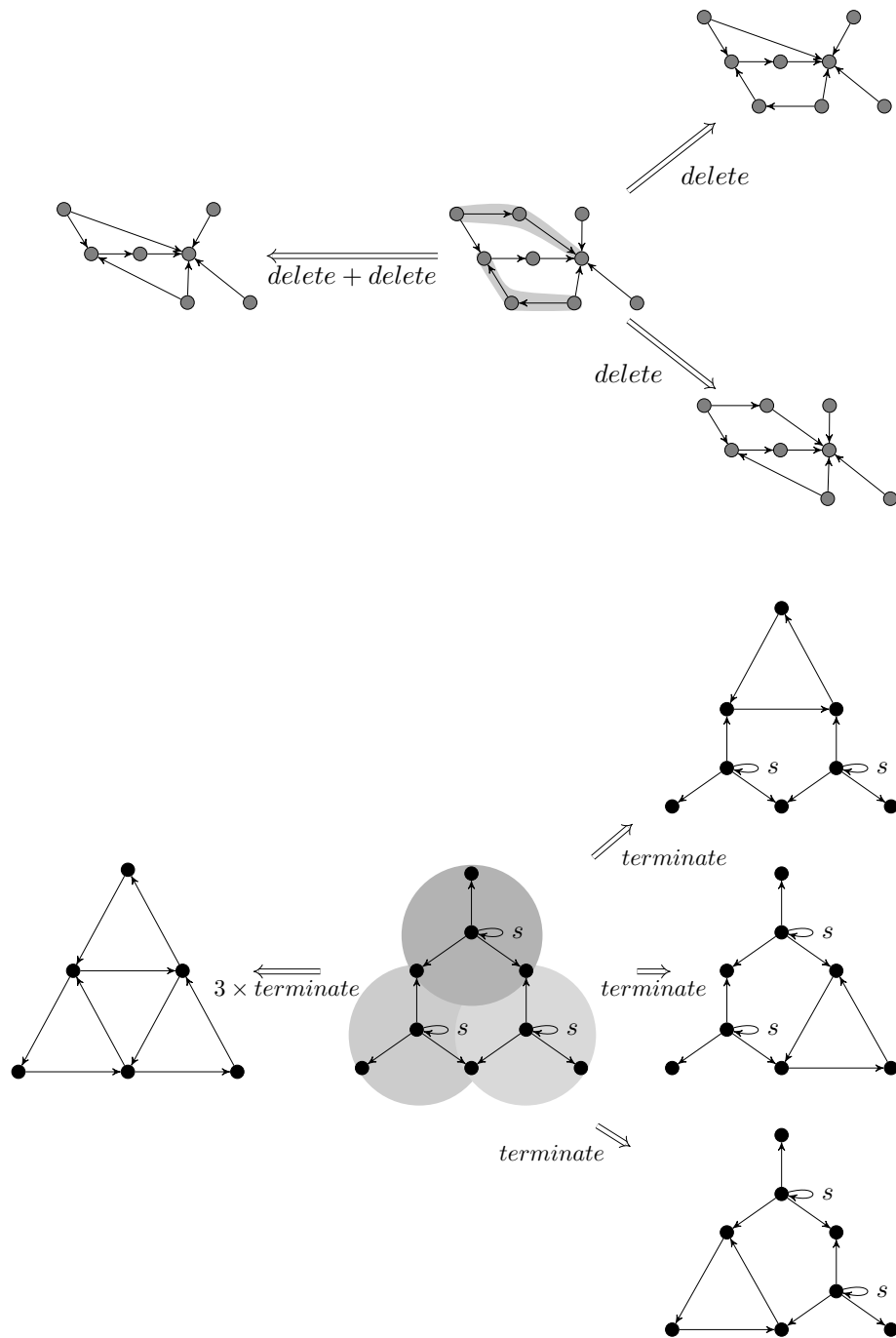


Figure 3.16: parallel independent direct derivations and their parallelization

3.7 Graph grammars and graph transformation units

3.7.1 Graph grammars

Similarly to other rule-based approaches, in graph transformation the notion of *graph grammar* proposes a system in order to generate specific languages. A graph grammar is given by an initial graph, a finite set of rules and a set of terminal symbols. A graph grammar specifies a language composed of all derivations starting from the initial graph and leading to graphs with labels from the terminal symbols.

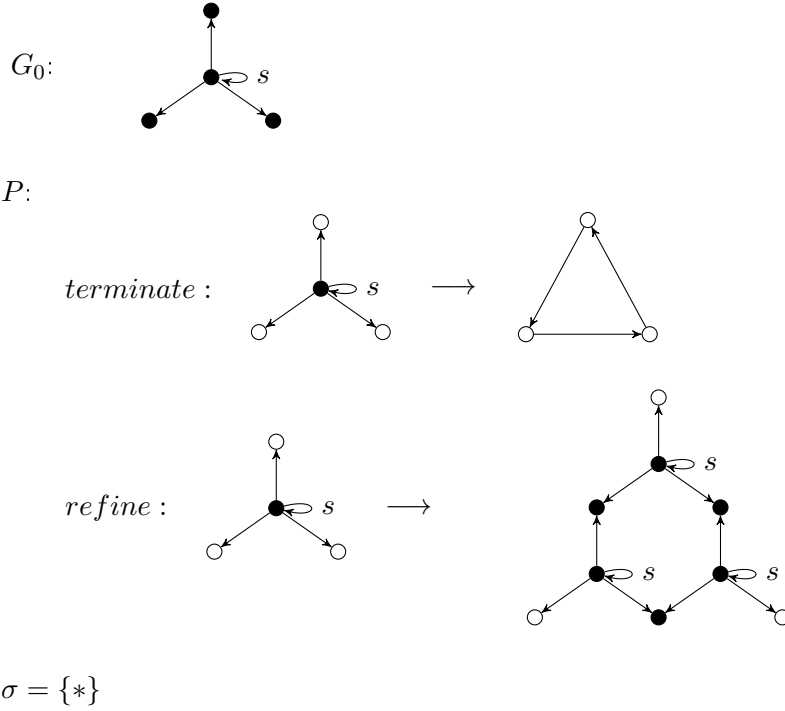
Definition 16 (graph grammar)

A *graph grammar* is a system $GG = (G_0, P, \Delta)$ such that $G_0 \in \mathcal{G}_\Sigma$ is the *initial graph*, P is a finite set of rules and $\Delta \subseteq \Sigma$ is a set of *terminal symbols*. GG specifies the language $L(GG) = \{G \in \mathcal{G}_\sigma \mid G_0 \xrightarrow[P]{*} G\}$ which consists of all graphs G labeled over Δ and which are derivable from the initial graph G_0 using rules in P .

Example 13

Let us consider the following graph grammar Δ -*grammar* $= (G_0, P, \Delta)$, where G_0 corresponds to the graph a tripod graph T_1 as introduced above. P is a set of two rules already introduced, *terminate* and the *refine* rule. Given a tripod graph the rule *refine* replaces it by a graph composed of three tripods connected with each other in such a way that if one replaces each one with a Δ graph, one gets a Δ^2 graph. The set of terminal symbols contains the unique symbol $*$ which means that only derivations leading to graphs without labels (in this case without the *s-loop*) belong to $L(\Delta$ -*Grammar*). The following Figure summarizes the components of Δ -*grammar*.

Sier-grammar:



It is obvious that the language of the grammar *Sier-grammar* contains Sierpinski graphs, but not only, because there is no guarantee that the refinement is done equally in each direction. Figure 3.17 displays an example of a graph in the *Sier-grammar* language.

3.7.2 Graph transformation units

The concept of *transformation units* has been introduced by Kreowski and Kuske [55]. It has been established during the last two decades as a modeling and structuring concept with several applications (see e.g., [61, 55, 56, 59, 58, 44, 57]). The concept can be considered as a generalization of graph grammars offering more flexibility in the specification of initial and terminal states by means of *graph class expressions*, but also permitting more control over the derivation process by means of *control conditions*. Graph transformation units can therefore be used for controlled computations. For more details about the expressive power as well as the theoretical background of graph transformational units see, e.g., [59, 62].

Control conditions

In order to regulate the graph transformation process restricting the non-determinism of rule application, control tools are needed. In the following, *control conditions* are introduced to fulfill this aim. A control condition permits the application of rules according to a given schemata. These can be based for example on their priority or by specifying a fixed sequence of rules that are applied sequentially. Another type of control conditions

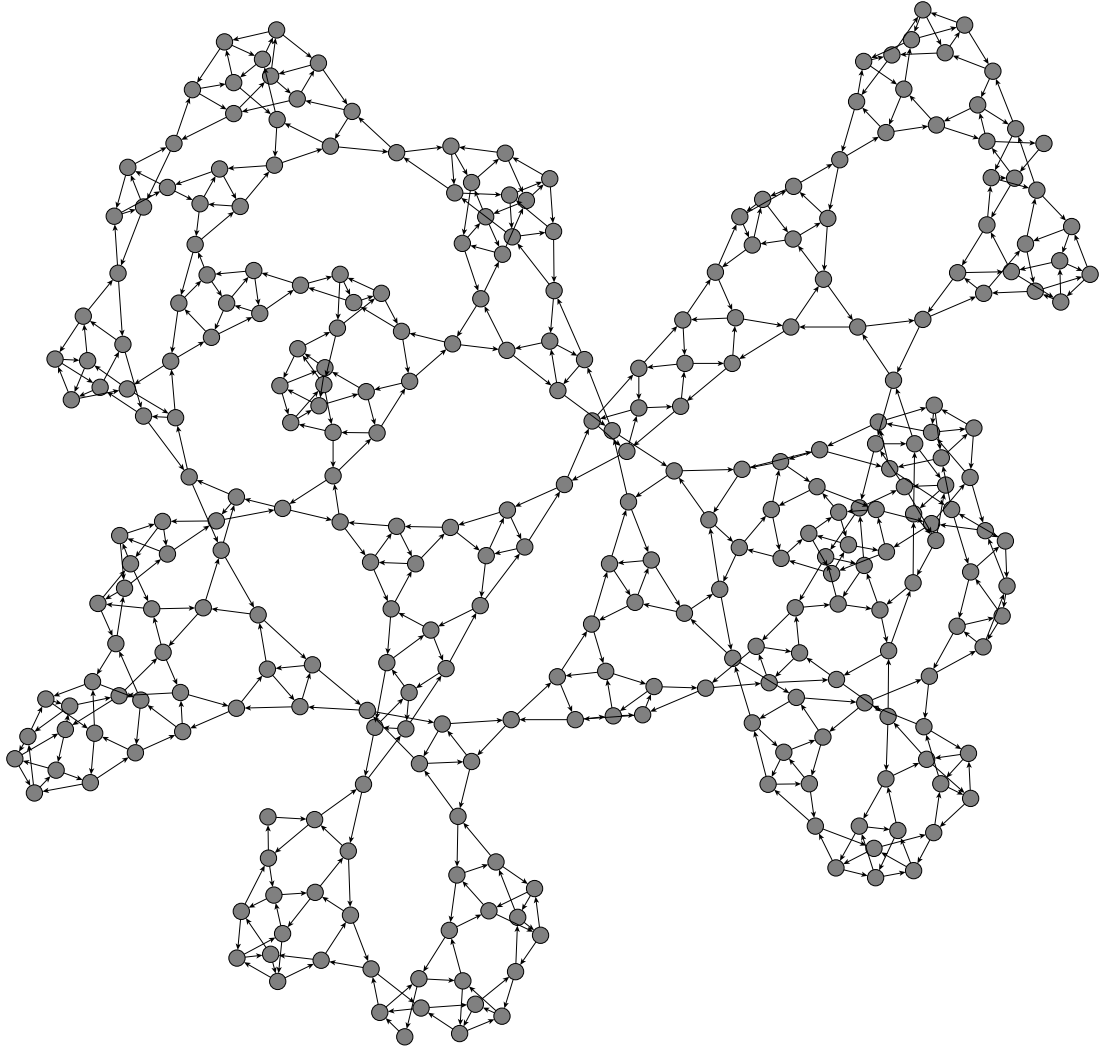


Figure 3.17: An example of a graph that belong to the *Sier-grammar* language.

restricts the matching process. It restricts the set of all matches of a given rules based on some constraints. More formally,

Definition 17 (control condition)

A *control condition* C over a set of identifiers ID is any syntactic entity over ID that specifies a set $SEM(C)$ of derivations.

Typical control conditions are regular expressions, priorities, as-long-as-possible and maximal parallelism.

1. Let P be a set of rules (names), regular expressions over P specify string languages over P . By definition, the constants \emptyset , $lambda$ and $r \in P$ are regular expressions and the composites $e_1; e_2$, $e_1|e_2$ and e^* are regular expressions if e_1, e_2, e are regular expressions. A derivation obeys a regular expression e if the application sequence of the derivation belongs to the language of e denoted $L(e)$. (i.e. $SEM(e) = \{G = G_0 \xRightarrow[r_1]{r_n} G_1 \cdots \xRightarrow[r_n]{r_n} G_n = G' \mid r_1 \cdots r_n \in L(e)\}$). In other words, $e_1; e_2$ allows a derivation if an initial section is allowed by e_1 and the remaining section by e_2 (i.e., $L(e_1; e_2) = L(e_1) \cdot L(e_2)$); $e_1|e_2$ allows a derivation if e_1 or e_2 allows it (i.e., $L(e_1|e_2) = L(e_1) \cup L(e_2)$); e^* allows a derivation if it is a sequence of sub-derivations each allowed by e (i.e., $L(e^*) = L(e)^*$). The expression $r \in P$ requires that r is applied (i.e., $L(r) = \{r\}$); $lambda$ allows any derivation of length 0 (i.e., $L(lambda) = \{id \mid G \xRightarrow{id} G'\}$); \emptyset forbids any derivation (i.e., $L(\emptyset) = \{\}$)
2. Given a regular expression e the notation $e!$ requires that e is applied *as long as possible*. In other words $e!$ specifies the set of all derivations $G \xRightarrow{*} G' \in SEM(e^*)$ such that there is no $G'' \in \mathcal{G}$ with $G' \xRightarrow{*} G'' \in SEM(e)$.
3. *Priorities* are another typical control condition. Let e_1 and e_2 be two regular expressions over P . The priority relation denoted by $e_2 > e_1$ requires that a derivation is allowed by e_1 only if there is no derivation allowed by e_2 .
4. *Maximal parallelism*, which is denoted by $||$, is a control condition that requires the application of a rule with maximum parallelism. In other words, given a rule $r \in P$, $||r||$ requires that all possible applications of r that can be performed in parallel (pairwise parallel independence according to Theorem 1) using r are performed in parallel.

Further examples of control conditions are introduced in the next chapter. The class of all control conditions is denoted by \mathcal{C} .

Control conditions can be composed by the operator $\&$ with $SEM(C_1 \& C_2) = SEM(C_1) \cap SEM(C_2)$ for all $C_1, C_2 \in \mathcal{C}$.

Graph class expression

Another important concept in graph transformation is the notion of a *graph class expression*. It allows the specification of a class of graphs. This is useful, for example, to define initial or terminal states of a graph transformational process.

Definition 18 (graph class expressions)

A *graph class expression* is any syntactic entity X that restricts the class \mathcal{G}_Σ to a subclass $SEM(X) \subseteq \mathcal{G}_\Sigma$. The class of all graph class expressions is denoted by \mathcal{X} .

- Each graph $G \in \mathcal{G}_\Sigma$ is a graph class expression with $SEM(G) = \{G\}$.
- Graph properties like *unlabeled* with $SEM(unlabeled) = \mathcal{G}_{\{*\}}$ or *simple* with $SEM(simple)$ is the set of all graphs having no multiple edges or loops.

Graph class expressions can be composed by the operator $\&$ with $SEM(X_1 \& X_2) = SEM(X_1) \cap SEM(X_2)$ for all $X_1, X_2 \in \mathcal{X}$.

In the following, graph transformation units are introduced.

Definition 19 (graph transformation unit)

A *graph transformation unit* is a system $gtu = (I, P, C, T)$ where I and T are graph class expressions that specify the *initial* and the *terminal* graphs respectively, $P \subseteq \mathcal{R}$ is a set of rules, and $C \in \mathcal{C}$ is a control condition over P .

The *semantics* of a graph transformation unit $gtu = (I, P, C, T)$ consists of all derivations $G \xrightarrow[P]{*} H$ allowed by C such that $G \in SEM(I)$ and $H \in SEM(T)$.

Example 14

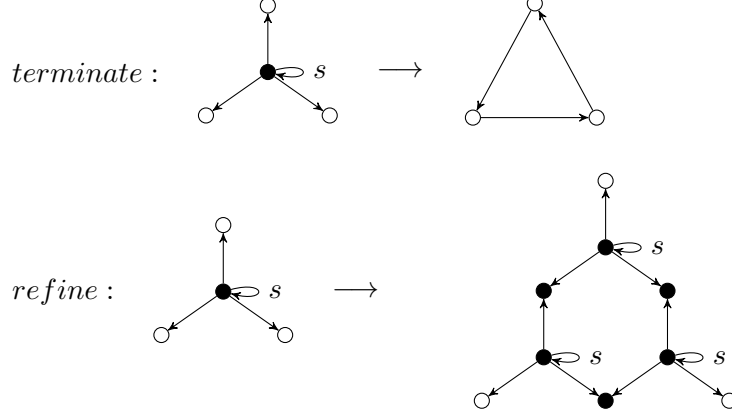
Figure 3.18 gives an example of a unit that produces Sierpinski triangles. The initial graphs are specified by the graph class expression $X\text{-graph}$. $SEM(X\text{-graph})$ contains all graphs that can be constructed recursively as follows: The tripod graph T as introduced above belongs to $SEM(X\text{-graph})$ and for each graph in $SEM(X\text{-graph})$ the result of the derivation using the rule *refine* (as introduced above) with maximal parallelism belongs to $SEM(X\text{-graph})$. Similar to the *Sier-grammar*, the set of rules P contains the two rules *terminate* and *refine*. The control condition requires that the rule *refine* is applied with maximum parallelism. This is repeated arbitrarily often. After that, the rule *terminate* is applied with maximum parallelism. The graph class expression *forbidden(loop)* requires that the terminal graphs have no loops.

It can be easily shown that the graph transformation unit *Sier-unit* generates Sierpinski triangles. Given a start graph $G_0 \in SEM(X\text{-graph})$, the application of *refine* with maximum parallelism divides all present tripods in the same time. Note that for each tripod in G_0 one can find six different matches of the rule *refine*. However, the Parallelization theorem requires that only one is chosen because they are not parallel independent. Otherwise, every two derivations having matches in two different tripods in G_0 are parallel independents. This includes of course the case where two matches have a common node, because it belongs to the image of the gluing graph of both matches. The refinement is repeated any number of times, after that the rule *terminate* is applied with maximal parallelism to change each present tripod in the current graph to a triangle. This is possible because similarly to *refine*, every two *terminate* derivations with two different tripods as matches are parallel independent. Figure 3.19 displays a Sierpinski triangle generated by the unit *Sier-unit* in seven steps. Starting with a graph composed by a

Sier-unit:

$I: X - \text{Graph}$

$P:$



$\mathcal{C}: ||\text{refine}||^* ; ||\text{terminate}||$

$T: \text{forbidden}(\text{loop})$

Figure 3.18: The graph transformation unit *Sier-unit*.

unique tripod the computation is performed by applying the rule *refine* with maximal parallelism six times. At the end the rule *terminate* is applied with maximal parallelism. For more examples of graph transformational units the reader is referred to the next chapters. Indeed graph transformation units play a very important role in the definition of *graph-transformational swarms*. To be more precise, an adapted version of graph transformation units are employed to specify the members of a swarm and their kinds.

3.8 Graph transformation tools

The aim of this section is to choose a graph transformational tool suitable for the purposes of this research work. A tool that permits, amongst others, to model parallel rule applications. It would be nice to also have structuring possibilities in order to model graph transformation units. A wide range of tools exists in the field of graph transformation. This section examines three of the most established tools. The Attributed Graph Grammar System (AGG), Graphs for Object-Oriented Verification (Groove) and Graph Rewrite Generator (GrGen.Net). All three tools use *typed attributed* graphs. This is an extension to graphs in such a way that the nodes and edges are from certain types and they can have additional information. It is more effective to search in a subset of items that have a given type instead of searching in all sets of nodes or edges based on labels

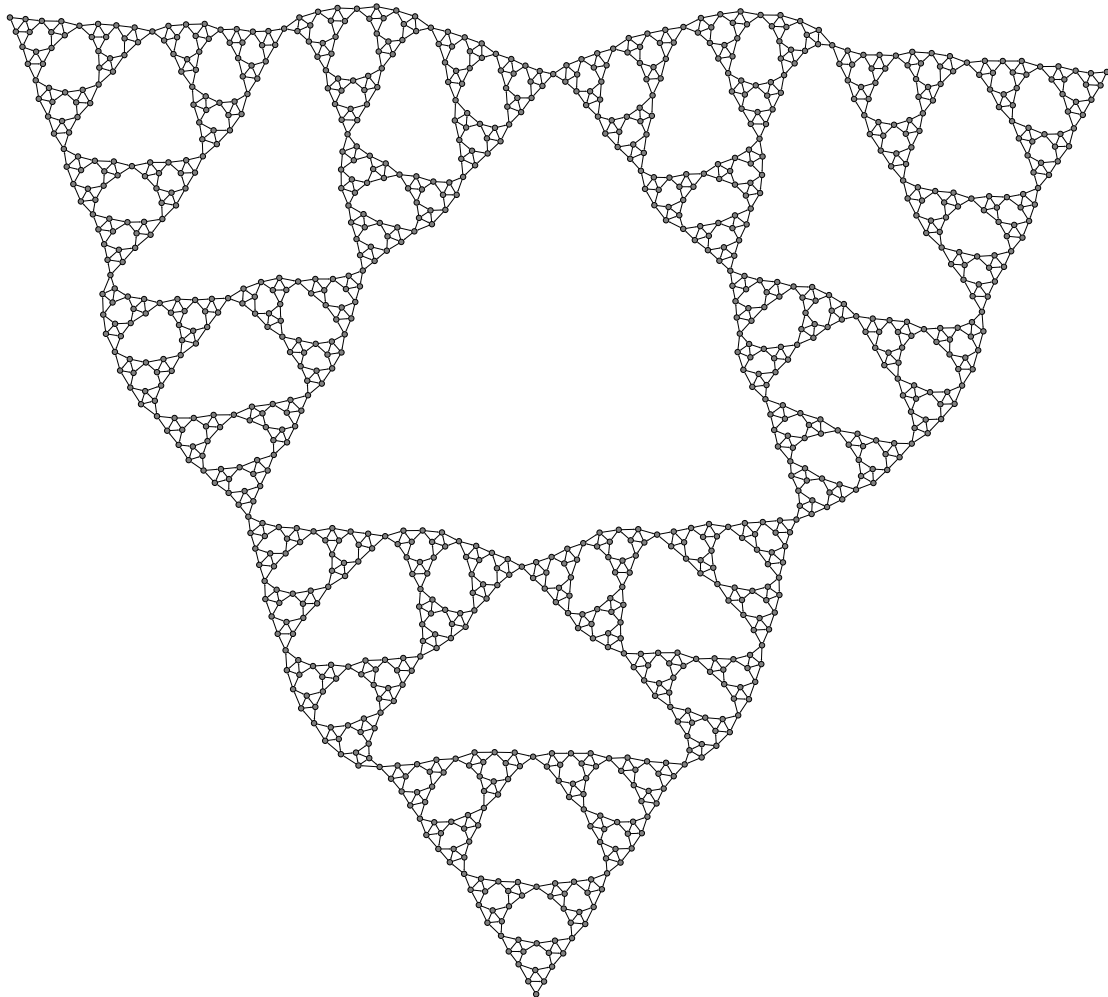


Figure 3.19: A Sierpinski triangle graph generated using the unit *Sier-unit*.

only. Furthermore, such a solution also makes it possible to visualize certain types with certain properties such as colors. For a formal introduction to typed attributed graphs see for example [30].

3.8.1 AGG: The attributed graph grammar system

AGG [96] is an environment tool for attributed graph transformation. It is implemented in Java and permits the utilization of Java objects for attributes and attributed conditions. This has a double effect. On one side, it allows one to easily define a wide range of computational solutions based on Java libraries and features, but it may also provoke arbitrary side effects because the Java semantics is not covered by a formal foundation. Every node and edge in a graph is associated with exactly one type from a given type set. The type is specified by a name and a graphical representation determining the shape, color etc. By default and internally, AGG uses the single-pushout approach. However, double-pushout is also supported. It is also possible to use negative application contexts. In addition to the manipulation of nodes and edges, the graph rules in AGG may perform attribute computations. Here it is also possible to call arbitrary Java methods in attribute expression.

The tool's environment provides a graphical interface for editing graphs and rules as well as for visualization of computation results. Moreover, it also provides an integrated textual editor for Java expressions. *AGG* system solves the problem of graph matching (i.e subgraph homomorphism problem) using a representation as a constraint satisfaction problem (CSP). It splits the solution algorithm from the graph model. This approach allows the use of different graph models without affecting the CSP solution algorithm. The application control in AGG is based on the notion of layers. Layers allow the specification of priorities between rules. Each layer contains a set of rules and has a priority. The computation process starts with layer 0 and applies all its rules as long as possible. Then it continues with the rules of layer 1, etc..

3.8.2 Groove: Graphs for object-oriented verification

Groove is a graph transformation tool that manipulates edge-labeled graphs without parallel edges. It is possible also to assign types to nodes and edges. It is based on the single-pushout approach with negative application conditions. However, it supports the double-pushout approach too.

Its main feature, compared to other graph transformation tools, and as the name already indicates, is its verification capability. Groove permits an exploration of the entire state space of reachable graphs. The user can choose different search strategies such as depth-first, breadth-first or linear. Moreover, Groove offers the possibility to verify properties specified in Computation Tree Logic (CTL) on the state spaces generated by a graph grammar.

Groove provides a precise control that specifies the allowed order of application of the rules in a given grammar in form of a *control program*. A control program specified by a language that contains, amongst others, looping, random choice between rules and simple function calls. In addition, the rules in Grooves have priorities which can be

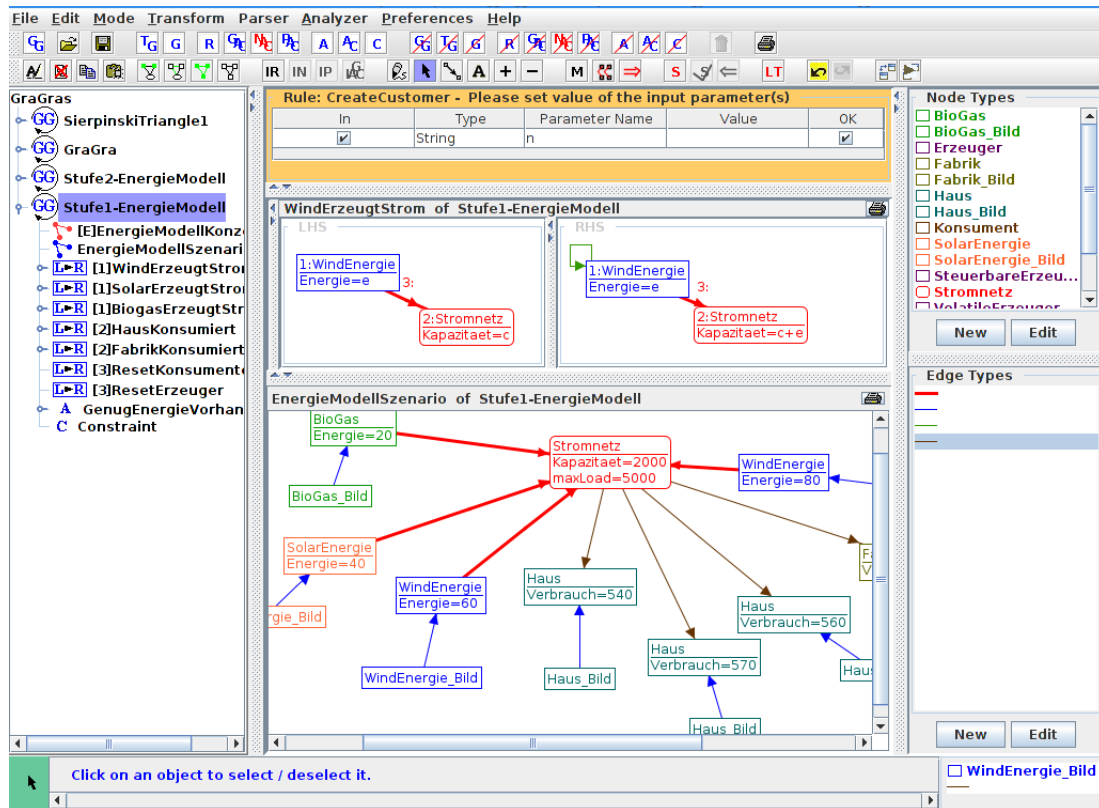


Figure 3.20: Screenshot of the AGG software.

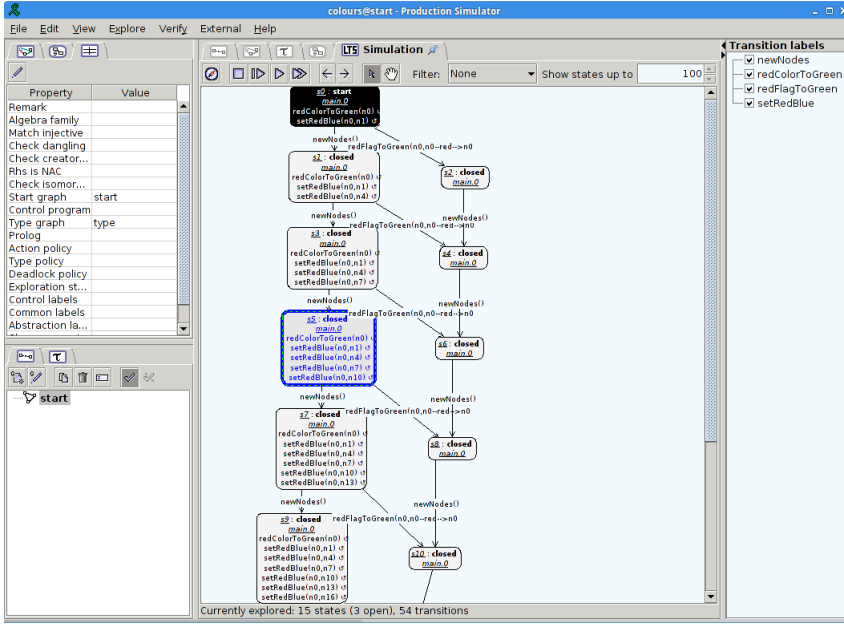


Figure 3.21: Screenshot of the Groove Simulator.

set as properties. The priorities also provide an additional scheduling control over rule applications. At the level of rules it is also possible to define sophisticated labeling that controls the matching behavior. To give an example, a label can be a regular expression over labels. Thus, an edge labeled by $l_1|l_2$ is matched if it matches either l_1 or l_2 .

Another feature of Groove is its graphical user interface which is called *the simulator*. It is implemented in Java with the aim to be useable by a large group of users. It allows graphical editing of rules and graphs and integrates additional functionality of a model checker with interactive exploration possibilities.

3.8.3 GrGen.Net: Graph rewrite generator

GrGen.Net is a graph transformation tool with a focus on execution efficiency. Based on Benchmark tests, it is indeed the fastest of the existing graph transformational tools (see, e.g., [37]). The efficiency of GrGen.Net is a result of the way the matching problem is represented and solved. The subgraph matching is performed according to a search plan allowing its expression as an optimization problem. The solution is then based on a heuristic method that employs knowledge over the host graphs. The implementation in .NET supports the rapid execution of GrGen.Net implemented programs.

The graphs in GrGen may be typed and attributed having directed as well as undirected edges. It is also possible to define multiple inheritances on node and edge types. Furthermore, GrGen.Net supports the specification of negative application conditions as well as sophisticated attribute computations over rules. The concept of graph transformation implemented in GrGen.NET follows the single-pushout approach. Alternatively, it supports the implementation of solutions based on the double-pushout approach.

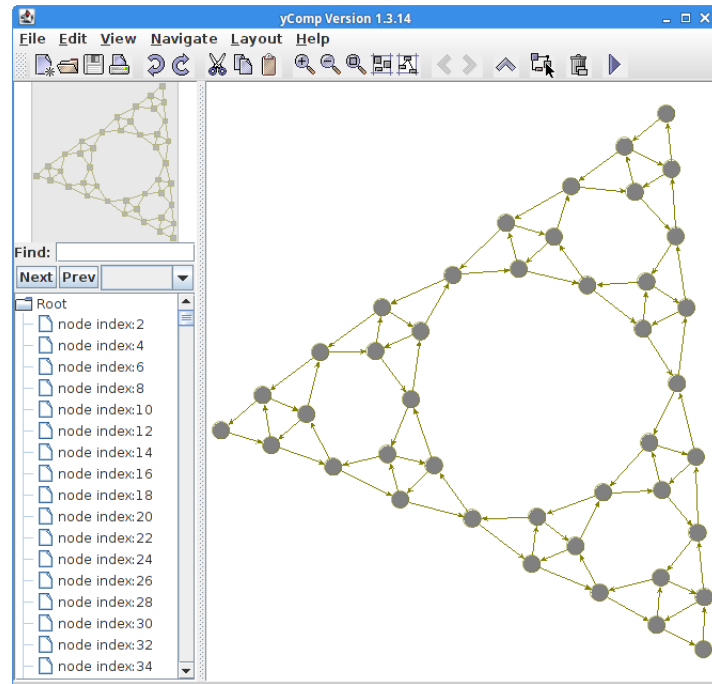


Figure 3.22: Screenshot of the graphical interface of GrGen.Net

Control over the application of rules is specified by so-called *regular graph rewrite sequences (RGS)*. RGS are synthetically related to regular expressions over rules including other expressions such as repetition of application as-long-as-possible or for a fixed number of steps. Another particularity of GrGen is the possibility to simulate simultaneous application of all matches of a rule.

The specification of graph grammars and application controls are based on script editing. However, it is possible to visualize the computation steps in a graphical interface. Figure 3.22 shows a screenshot displaying the result of a computation step of the graph transformation unit *Sier-unit*.

3.8.4 Comparison between the three tools

All three tools offer a wide range of utilities. However, within the scope of this work, parallelism plays a very important role. GrGen.Net is the only tool that allows to at least simulate parallelism. Given a rule r , it is possible in GrGen.Net to apply this rule with simulated maximum parallelism using a specific control condition as follows: The underlying engine searches for all matches of r first and then applies the rule sequentially using those matches. Moreover, the syntax of the control language in GrGen.Net seems

to bear the greatest similarity to that of graph transformation units. Another advantage of GrGen.Net compared to the two other tools is the possibility to use rules as structuring devices. A rule can call other rules to be executed following a control condition. All these factors contributed to choosing GrGen.Net as the primary tool in which the developed solutions in this thesis are implemented. The next section introduces the first example of an implementation showing the advantages of the tool as well as the problems occurring regarding parallelism.

3.9 First implementations

This section starts the discussion of the possibility to implement the components introduced in this thesis in GrGen.net focusing on the control condition maximal parallelism. Other components are discussed in the next chapters.

The unit *Sier – unit* is taken as illustrative example. The codes in Listing 3.1 are the direct translation of the rules *refine* and *terminate* in Figure 3.18. The lines between 2 and 5 and between 17 and 20 specify the left-hand side of each rule. For example, Line 17 stands for a node named *s* that has an outgoing edge called *one* with a target *x*. Lines 18 and 19 complete the tripod construction. Line 20 requires that the *s*-node has a loop. After the keyword *modify*, the modification that should be performed on the underlying match is described. Two edges are deleted, and 5 new nodes are created. *s1* and *s2* represent the centers of the new tripods. The nodes are connected with each other to construct an *X*-graph composed of three tripods as in the right hand side of the rule *refine*.

The translation of the control condition of *Sier-unit* is realized partially as displayed in Listing 3.4. Partially because there is not a direct expression of arbitrary repetition in GrGen.Net. For that, and for the first attempts, fixed numbers of repetitions are chosen to perform the first experiments. In this example, 6 repetitions of maximal parallel application of *refine* is allowed. Maximal parallelism is translated using the operator "[]" in GrGen.Net. The expression *[r]* in GrGen.Net, called maximal matching, requires that all possible matches of *r* are chosen and then the rule is applied (sequentially) with respect to all these matches.

The problem here is that two derivations can be applied without being parallel independents. To illustrate this effect, let us consider the application of the rule *refine* with maximal matching in *GrGen.Net* to an initial graph consisting of a unique *T*-graph. Figure 3.23 shows the result of this application. The figure also shows how edges and nodes are matched in the start graph. This is one of the useful features of the GrGen.Net tool. As expected, there are six possible matches and the execution does not take into consideration the parallel independence of derivations.

Listing 3.1 The rules of the unit *Sier-unit*

```

1: rule terminate1(){
2:   s:Node-one:Edge->x:Node;
3:   s-two:Edge->y:Node;
4:   s-three:Edge->z:Node;
5:   s-loop:Edge->s;
6:   modify{
7:     delete(one);
8:     delete(two);
9:     delete(three);
10:    delete(s);
11:    y-->x;
12:    x-->z-->y;
13:  }
14: }
15:
16: rule refine1(){
17:   s:Node-one:Edge->x:Node;
18:   s-two:Edge->y:Node;
19:   s-three:Edge->z:Node;
20:   s-->s;
21:   modify{
22:     delete(two);
23:     delete(three);
24:     s-->a:Node;
25:     s-->b:Node;
26:     s1:Node-->a;
27:     s1-->y;
28:     s1-->c:Node;
29:     s1-->s1;
30:     s2:Node-->b;
31:     s2-->c;
32:     s2-->z;
33:     s2-->s2;
34:   }
35: }

```

Listing 3.2 The control condition (control1) in GrGen.Net *Sier*-unit

```

1: debug exec(init;> [refine1][6]);> [terminate1])

```

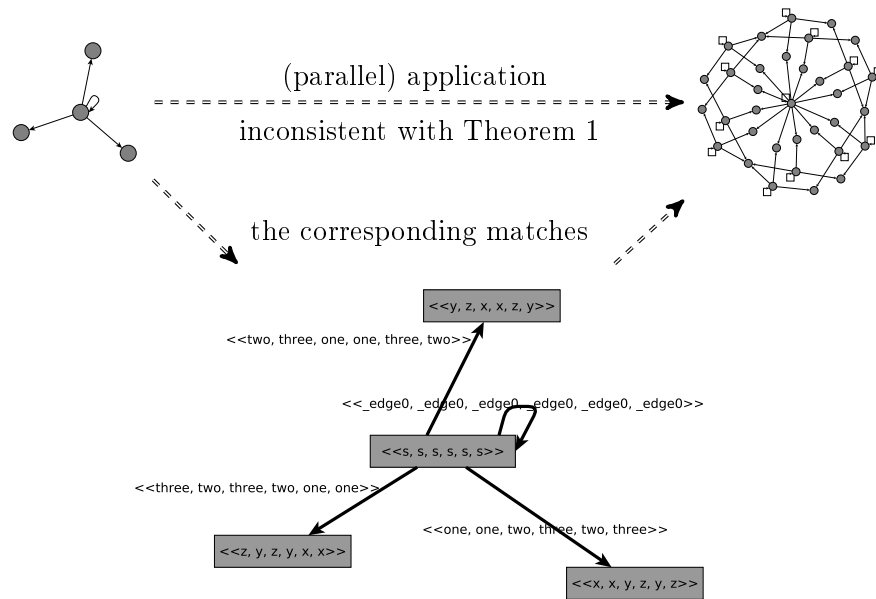


Figure 3.23 Derivation using the simulated maximal parallelism of the rule *refine* in GrGen.Net without adaptation.

To overcome this unwanted effect, the notion of edge classes in GrGen.Net is used¹. Three edge classes are defined *One*, *Two* and *Three*. The tripod subgraphs are adapted such that each outgoing edge up to the loop belongs to a different class. In this way, for the rules *refine* and *terminate* there is one possible match in each *T*-subgraph. The adaptation is in Listings 3.3 and 3.4. Figure 3.24 gives the output of GrGen.Net after adaptation for the first two steps. In each step the maximal parallel application of rule *refine* is truly simulated.

Listing 3.3 The rules of the unit *Sier*-unit with adaptation

```

1: rule terminate(){
2:   s:Node-one:One->x:Node;
3:   s-two:Two->y:Node;
4:   s-three:Three->z:Node;
5:   s-loop:Edge->s;
6:   modify{
7:     delete(one);
8:     delete(two);
9:     delete(three);
10:    delete(s);
11:    y-->x;
12:    x-->z-->y;
13:  }
14: }
15:
16: rule refine(){
17:   s:Node-one:One->x:Node;
18:   s-two:Two->y:Node;
19:   s-three:Three->z:Node;
20:   s-->s;
21:   modify{
22:     delete(two);
23:     delete(three);
24:     s-:Two->a:Node;
25:     s-:Three->b:Node;
26:     s1:Node-:One->a;
27:     s1-:Two->y;
28:     s1-:Three->c:Node;
29:     s1-->s1;
30:     s2:Node-one2:One->b;
31:     s2-:Two->c;
32:     s2-:Three->z;
33:     s2-->s2;
34:   }
35: }
```

Listing 3.4 Control *Sier*-unit with adaptation

¹Note that it is also possible to choose labeling as adaptation strategy.

```

1: debug exec(init;>
2: [refine][6]);>
3: [terminate])

```

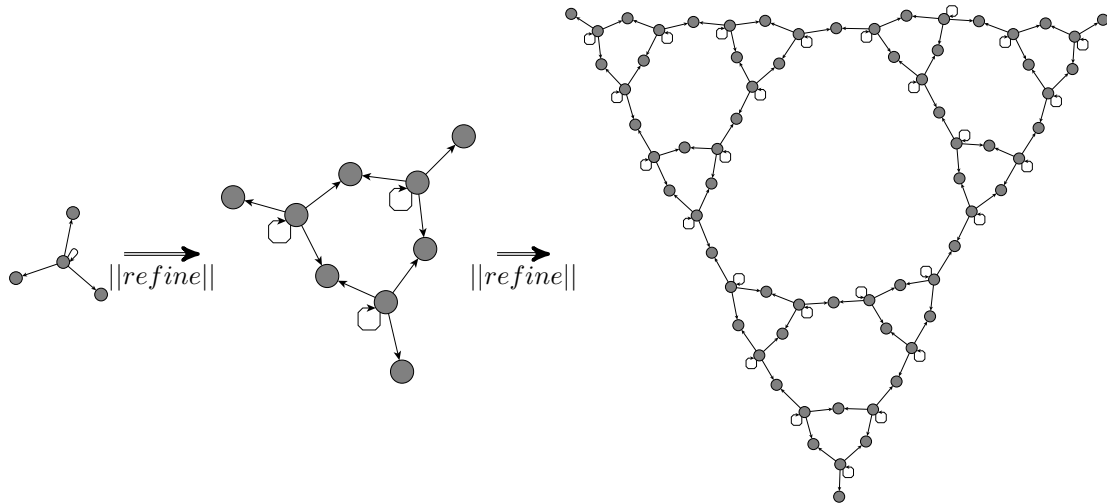


Figure 3.24 A sample computation of *Sier-unit*

3.10 Summary

This chapter has presented an overview of graph transformation providing the basic elements and methods needed in the current research work. In the chosen framework, graphs are directed edge-labeled. The transformation of graphs is performed based on double-push-out rule applications. In order to extend the modeling flexibility, positive and negative contexts are allowed. They restrict the applicability of a given rule based on surrounding items in the underlying match.

One of the characteristics of graph transformation is the systematic formulation of parallelism of two derivations by means of the parallelization theorem. This chapter has proposed a generalization of this theorem for a set of derivations. This can also be considered as one of the main contributions of this research work.

In order to regulate the rule applications graph transformation units are used as structuring concept. A graph transformation unit is a computational device that contains a set of rules, a control condition and two graph class expressions that specify the initial and terminal class of graphs.

Three software tools have been compared to determine which one is more suitable for the purposes of this work. GrGen.Net seems to be more suitable regarding parallelism.

However, it should be used carefully because the engine does not check the parallel independence of derivations.

This chapter started to explore how to implement the introduced components in Gr-Gen.Net. The implementation of the control condition maximal parallelism was illustrated and discussed.

Chapter 4

Graph-transformational swarms

This chapter introduces *graph-transformational swarms* as a novel approach to swarm computation. Chapter 2 introduced swarm computing by means of the inspiration from nature and the major existing methods. Chapter 3 provided a graph-transformational approach where the rules can be applied in parallel. Graph-transformational swarms combines the knowledge of the two previous chapters. It is based on the ideas of swarms and swarm computing and is formulated using the capabilities of graph transformation. The key is that the framework of graph transformation provides the concept of parallel rule application to formalize the simultaneous actions of swarm members.

A graph transformational swarm consists of members all of the same kind or of different kinds where the number of members is fixed by a given size. Kinds and members are modeled as graph transformation units that consist of a set of graph transformation rules specifying the capability of members and a control condition which regulates the application of rules. The members of a swarm act on an environment, which is represented by a graph, by applying their rules.

This chapter is structured as follows. Section 4.1 summarizes and combines the main ideas from the previous chapters in order to define the components needed for graph-transformational swarms. Section 4.2 provides a formal definition of graph-transformational swarms and their computation. In Section 4.3 two examples are given to illustrate the concept: a very simple ant colony and a swarm that computes Hamiltonian cycles. Section 4.4 extends the notion of control conditions in order to specify the stochastic matching of rules. The way how to use the developed framework in practice is discussed in Section 4.5. It studies the modeling and the implementation of the components of graph-transformational swarms using different concepts. Finally, a summary is provided in Section 4.6.

4.1 The main ideas of swarms

This section combines the ideas and notions described in the two previous chapters summarizing the main components needed to introduce graph-transformational swarms in

the next section. The methodical approach comprises (1) the knowledge gained from observations of swarms in nature, (2) the analysis of the major swarm computing methods and (3) the formal methods of graph transformation. Thus, the following description of the main ideas conforms to this approach. For each component, first the biological interpretation is given, followed by the solutions proposed in the major swarm computing approaches ant colony optimization (ACO), particle swarm optimization (PSO) and cellular automata (CA), and finally, the corresponding graph-transformational formulation is proposed. The description is accompanied by illustrations using the components of a simple ant colony example, which models ants that walk in a graph searching for *food*. The details of this example are introduced incrementally in this section.

4.1.1 Environment

The environment plays an important role in swarm behavior. It is used for example for indirect communication between the swarm members. This is very obvious in the foraging behavior of ant colonies. Ants use pheromone, which they drop down on the surface to communicate with each other in a colony. In Bird flocks, the individuals which act in the air can keep a permanent eye-contact with their neighbors. In the medium water, which is the environment of fish schools, visualization is more difficult. However, water facilitates the transmission of vibration, making it possible for the individuals in the school to perceive the movement of other fishes or external individuals or obstacles in the neighborhood.

In swarm computing, such environments are modeled in order to play similar roles. In ACO for example, the environment which is also called the search space is modeled as a graph where "artificial" pheromone values are assigned to the edges. These values are updated during the computing process. In PSO and CA, the environment is encoded using data structures that permit a specification, as well as an easy update of the neighborhood which is changed during the computing process. ACO is the concept that is most consistent with the ideas of graph transformation, since in ACO the environment also corresponds to a graph. The question that should be handled in this research work is how to encode the environments of other swarm computing approaches where usually graphs are not explicitly used as data structure. This work is based on the assumption that graphs are powerful enough to permit such a generalization, offering several advantages of the same time. In Chapter 5 it will be demonstrated how graphs can be used to represent the environment in PSO and CA.

The underlying framework uses the notion of graph class expression as introduced in previous chapter (precisely, on page 52) to specify the environment. Roughly spoken, a graph class expression is any expression that specifies a class of graphs. Thus it is possible to define not only a unique graph but also to specify a whole class of graphs that a swarm can have as start graph "initial environments". Furthermore, graph class expressions are used to specify the goal of a swarm. In addition to the examples already given, the following typical examples of graph class expressions are used in the context of swarms:

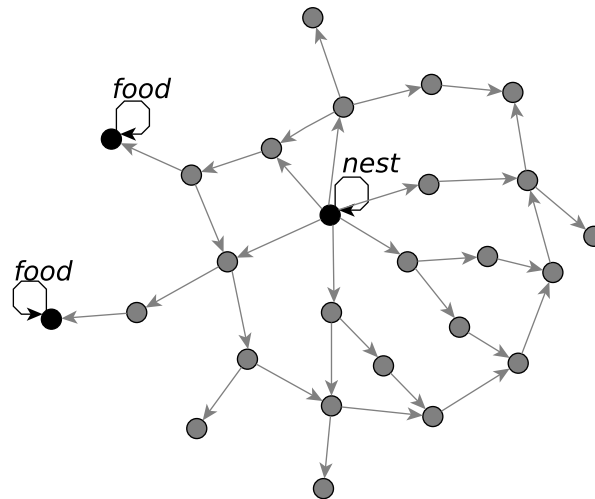


Figure 4.1 A graph example in the graph class specified by the graph class expression $\text{some-}\{\text{nest}, \text{food}\}\text{-looping}((\text{unlabeled} \ \& \ \text{simple}))$

- $\text{SEM}(\text{required}(X))$ for $X \in \mathcal{X}$ contains all graphs with a subgraph in $\text{SEM}(X)$.
- $\text{SEM}(\text{op}(X))$ for some graph operator op and $X \in \mathcal{X}$ contains all graphs obtained by the application of the operator to graphs in $\text{SEM}(X)$. Explicit examples of such operators are $\text{all-}\Delta\text{-looping}$, $\text{some-}\Delta\text{-looping}$ for $\Delta \subseteq \Sigma$ and $\text{some-}\{\text{nest}, \text{food}\}\text{-looping}$. Applied to $G \in \mathcal{G}_\Sigma$, the first operator adds exactly one δ -loop for some $\delta \in \Delta$ to each node of G , the second operator adds exactly one δ -loop to some nodes and the third operator adds some nest -loops and some food -loops to some nodes. Figure 4.1 displays an example of a graph in $\text{SEM}(\text{some-}\{\text{nest}, \text{food}\}\text{-looping}((\text{unlabeled} \ \& \ \text{simple})))$.
- $\text{SEM}((\text{forbidden}(H)))$ for $H \in \mathcal{G}$ contains all graphs without a subgraph isomorphic to H .

4.1.2 Rules

As a direct consequence of the self-organization theory (Section 2.1), which is established to explain swarm behavior in nature, the members of a given swarm follow a very limited number of rules. Furthermore, the rules are based on local information. That is, none of the members have an overview of the whole swarm and there is no entity that controls the behavior of the swarm centrally. However the behavior of animals cannot be fully explained by the strict compliance to such local rules. For example, in ant colonies, the rule to choose the paths with the best pheromone value is mostly heeded by the ants but not always. Similarly to other natural phenomena, scientists consider this behavior as a stochastic process and has therefore been formalized using parametrized probabilistic functions. That is, functions that contain noise terms and can be adapted

to generate results fitting a given model. In many studies, the simulated behaviors are astonishingly good. They statistically correspond to the original behavior at least within a specific experimental setup (see Section 2.1). In summary, the rules of swarms in nature exhibit two important aspects, namely the locality principle and non-determinism.

The rules in swarm computing approaches are related to this and accordingly share two characteristics. First, there is a kind of non-determinism expressed in the computational models by probabilistic functions. This is very important in terms of optimization purposes. The generated noise promotes the exploration of the search space and permits the avoidance of local optima. The second characteristic is the locality aspect of rules. That is, the rules also use local information from the local region in the environment (such as in ACO) or from the neighbors (such as in CA and in PSO).

The rules and their applications in graph transformation exhibit properties that correspond with the required characteristics of swarms and swarm computing. The rules in graph transformation are specified by a left hand side graph L , a gluing graph K and a right hand side graph R , such that K is a common subgraph of L and R . The first step of the application of a rule to a given graph G consists of finding a match of L in G . The good thing here regarding the swarms requirement is that the matching process is non-deterministic. However, in some cases something more than arbitrary choice is needed. In order to model stochastic processes, control tools based on probabilistic calculations are required. As described in the previous chapter, control conditions are a well suited method to break the non-determinism in rule application. The question now is: is it possible to develop control conditions that produce stochastic behavior in the chosen graph transformation approach? Section 4.4 answers this question proposing such kind of control conditions. To continue our description and to show the locality principle when applying graph transformational rules, let us consider the rule in Figure 4.2 which is called *food*. It is used along with other rules for modeling a simple ant colony. The

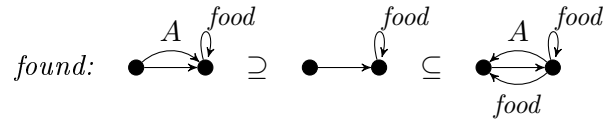


Figure 4.2 A graph transformation rule that models the behavior of an ant A when it finds *food*

rule *food* models an ant A that finds *food* and returns back by reversing the direction of A -edge and creating a parallel *food* edge. This example shows that given an ant A the application of the rule depends only on the information in the neighboring edges and nodes, namely in this case a node with a *food*-loop. The application leads to changes solely of the neighborhood. Here the edge labeled by A is reversed and a parallel edge is created. In general, the rules that will be used in swarms follow this schemata. The members of swarms are assigned to an edge or a node or more generally a small subgraph. The local information corresponds to the state of the neighboring edges and nodes and

the local changes consist of creating or deleting neighboring edges or nodes.

In this way the locality principle is satisfied in an elegant manner. Note that all these considerations hold true for rules with application contexts (as introduced in 3.5), since the application context can be very well specified in a predefined neighboring region of the left hand side of a rule.

4.1.3 Kinds and members

Another important component in swarms is the way in which the individuals are organized. As described in Section 2.1, an ant colony is usually organized in subgroups: workers, drones and queens. Every subgroup has predefined tasks in the swarm. Many other insects that exhibit swarm behavior, such as bees, wasps and termites, are organized in similar structures.

The description of the structural organization of insects above gives the impression that the computing methods inspired by these should also offer the possibility to define subgroups. However, this is not the case, at least in the major swarm computing approaches. ACO for instance is inspired by the foraging behavior of a colony. In this behavior, only one subgroup is directly involved, namely the subgroup of workers.

Despite of the prevailing approach which assumes that all members should have the same role in a swarm, it is decided in the present thesis to offer the possibility that members can have different tasks. In addition to being inspired by swarms in nature as mentioned above, this choice is also motivated by the current trend in the application of swarm computing. It appears to be very promising to combine one swarm computing method with other swarm computing methods or with other computing approaches such as fuzzy logic, evolutionary algorithm or artificial neural networks to solve complex problems. The resulting solutions are also known as *Hybrid swarm algorithms* (see for example [68]). These sort of solutions can be interpreted as a swarm composed of members that have different tasks. Every involved method specifies a kind of task.

The notion of *Kind* is introduced below in order to reflect this idea.

A kind consists of a simple graph transformation unit which is an adapted version of the graph transformation units introduced in the last chapter. Simple transformation units are graph transformational units without initial and terminal specifications.

Definition 20 ((simple) graph transformation unit)

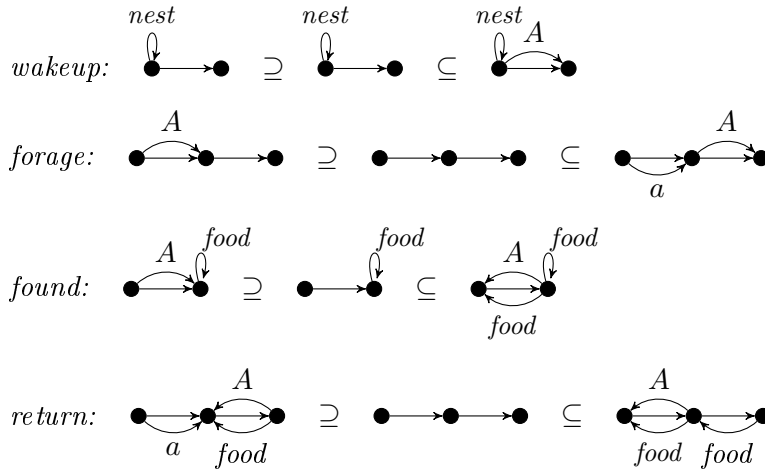
A (simple) *graph transformation unit* is a pair $gtu = (P, C)$ where $P \subseteq \mathcal{R}$ is a set of rules, and $C \in \mathcal{C}$ is a control condition over P .

Section 3.7.1 has introduced regular expressions, as-long-as-possible and priorities as typical control conditions. Here an additional control condition which is needed in the context of graph transformational swarms is given. *free* is a control condition that allows all derivations using the rules in P without restrictions. This control condition is particularly useful for inducing transformation units from rules. More precisely, each set $P \subseteq \mathcal{R}$ of rules induces a graph transformation unit specified by $gtu(P) = (P, free)$. For $gtu(\{p\})$ with $p \in \mathcal{R}$ we write $gtu(p)$ for short.

Figure 4.3 illustrates the notion of kind. The kind *ant* is specified by a unit composed of four rules and a control condition. The ant *A* wakes up by accessing edges that exit nodes with *nest*-loops. Then it forages for food by passing one edge and marking the old position. If *A* reaches a node with a *food*-loop, it starts to return, retraveling along its own marked path while marking this path with the label *food*. The control condition requires that the ant *A* wakes up at the beginning, but never again. Moreover, the application of the *return*-rule has higher priority than the *found*-rule, which in turn has higher priority than the *forage*-rule. Hence, the ant stops foraging whenever it finds food. From there on it returns along the path labeled with *a* because the return rule has the highest priority.

ant

rules:



control: $(wakeup ; (forage \mid found \mid return)^*) \ \& \ (return > found > forage)$

Figure 4.3 The kind *ant*

Given a defined kind, a set of *related* members can be generated by relabeling.

Definition 21 (related unit)

A unit *gtu* is *related* to a unit *gtu*₀ if *gtu* is obtained from *gtu*₀ by relabeling. For a mapping $rel: \Sigma \rightarrow \Sigma$, the relabeling of *gtu*₀ is the unit $rel(gtu_0) = (rel(P_0), rel(C_0))$ where the relabeling replaces each occurring $x \in \Sigma$ in the components *P*₀ and *C*₀ of *gtu*₀ by $rel(x)$. The set of units related to *gtu*₀ is denoted by $RU(gtu_0)$.

That is, given a kind *k* specified by a unit *gtu*₀ one can generate an arbitrary number of members¹ in $RU(gtu_0)$. For example, one can create $n \in \mathbb{N}_{>0}$ members of kind *ant* each called *ant*_{*i*} for $i \in [n]$. Such that *ant*_{*i*} is obtained by replacing each occurrence of *A* and *a* respectively by *A*_{*i*} and *a*_{*i*}. When designing a kind, it is important to determine which

¹The size of a kind will be introduced more formally in the next section

labels are specific for the members and which are not in early stage. Generally, the label that specifies the name of a kind is relabeled (in our example, this corresponds to the label A). The relabeling or not relabeling of created items can have crucial consequences for the cooperation between the members. In our illustrative example, if the marks a are not relabeled, then an ant in the swarm follows not only its own created path when returning but follows any other path from any other ant in the swarm.

4.1.4 Parallelism

A second consequence of the self organization theory is that swarm members in nature act simultaneously. The animals are active all the time and there is no waiting for input from other members or other entities.

In ACO and PSO this characteristic, which in computer science terminology can be termed as parallelism, is not systematically specified. Although there is quite a lot of research that proposes parallel solutions based on both approaches. The proposed solution approaches are specific for a given problem and lack general methodology (For a general overview of the parallelization strategies of ACO and PSO see, e.g., [75, 105]) In CA, parallelism is simple but well specified. All cells act in parallel reacting on neighboring information. The parallelism in CA is responsible for the emergent behavior that is subject to research in CA theory.

Parallelism in our approach is specified by a formal semantics. Its description is therefore precise and unambiguous. The specification is provided by the generalized parallelization theorem as introduced in the last chapter. Amongst other things, the theorem specifies when two or more rules can be applied in parallel. It is therefore possible to model distributed systems and determine how their computations can be simulated sequentially.

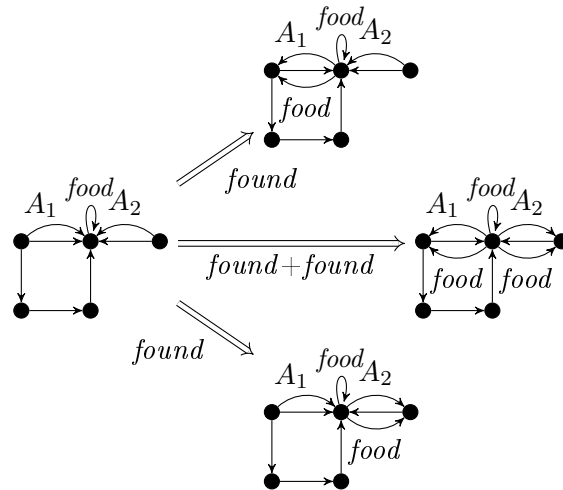


Figure 4.4 parallel application of the rule *found* by two different ants

To illustrate this, let us consider again the example of the rule *found* but this time performed by two different ants A_1 and A_2 in parallel. Lets assume that the two ants are both in front of *food-loop* i.e., both can apply their *found* rules. Figure 4.4 displays such a situation. Because the rules change only the edges labeled by the names of the ants, each ant can apply its own *found* rule in parallel. If in contrast it is assumed that the rule deletes the *food-loop*, then the parallel application in this situation is not possible. That is, only one member can apply its rule *found*. Due to the explicit specification of such situations, there is an obligation to deal with situations where it is possible that members can enter conflicts. It will be shown how it is possible to deal with such situations based on different examples. Furthermore, later chapters will propose distributed versions of ant colony optimization and particle swarm optimization developed in a natural way.

4.1.5 Cooperation

Referring again to self-organization theory, something like cooperation does not exist in swarms in nature. Or at least there is not an explicit cooperation from the point of view of the individuals in a swarm.

In ACO and PSO, as well as in other heuristic optimization methods in general, the search for optimal (or good enough) solutions consists of the repetition of certain procedures until a terminal condition is met. This repetition can alternate with some adaptations based on additional local search methods. After each repetition the solutions found so far are evaluated and the best solutions are updated. This regulation of the computation process as well as the alternation between different kinds of procedures can be considered as a sort of cooperation between different procedures.

The generalization approach offers the possibility to formulate such a cooperation. For that, the notion of *cooperation conditions* is employed.

A *cooperation condition* is any control condition over kinds. It specifies when members of a given kind are allowed to apply their rules. Typical cooperation conditions are introduced in the next section.

Table 4.1 summarizes the main ideas of swarms and swarm computing and their interpretation based on a graph-transformational view. A swarm acts in an environment (graph), starting in an initial one (graph class expression). A swarm consists of members (related units) of some kinds (graph transformation units). The members follow simple rules (graph transformational rules), interact simultaneously (parallel rule application) and in a coordinated way (control condition). A swarm may have a goal (graph class expression)

main components of swarms and swarm computing	graph transformational interpretation
environment	graph
initial environment	graph class expression
rules	graph transformation rules
kinds	graph transformation units
members	related units
simultaneous action	parallelism
cooperation	cooperation condition
goal	graph class expression

Table 4.1 The main components of swarms and swarm computing and their graph transformation counterparts

4.2 Graph-transformational swarms and their computations

This section introduces graph-transformational swarms and their computations. The swarm members act simultaneously in a common environment represented by a graph. All the members of a swarm may be of the same kind or of different kinds to distinguish between different roles members may play. The number of members of each kind is given by the size of the kind. To increase the flexibility of this notion, multidimensional swarms are also allowed by means of size vectors. In this case, the number of members of the respective kind is the product of the size components. Given a size vector $(n_1, \dots, n_l) \in \mathbb{N}_{>0}^l$, the index vectors (i_1, \dots, i_l) with $i_j \in [n_j]$ for $j \in [l]$ are used to identify the members of the swarms.² While a kind is specified as a simple graph transformation unit, the members of a kind are modeled as units related to the unit of this kind making sure in this way that all members of some kind are alike. A swarm computation starts with an initial environment and consists of iterated rule applications. It requires massive parallelism meaning that each member of the swarm applies one of its rules in every step. In other words, each member acts sequentially according to its specification while all of them together are always busy. The choice of rules depends on their applicability and the control condition of the members. In some cases, a more restricted way of computation is reasonable. Hence, it is allowed to provide a swarm with an additional cooperation condition. Finally, a swarm may have a goal given by a graph class expression. A computation is considered to be successful if an environment is reached that meets the goal.

Definition 22 (swarm)

A *swarm* is a system $S = (in, K, size, M, coop, goal)$ where *in* is a graph class expression specifying the set of *initial environments*, K is a finite set of graph transformation units,

² $\mathbb{N}_{>0} = \mathbb{N} - \{0\}$ and $[n] = \{1, \dots, n\}$.

called *kinds*, *size* associates a *size* vector $size(k) \in \mathbb{N}_{>0}^{d(k)}$ with each kind $k \in K$ where $d(k) \in \mathbb{N}_{>0}$ denotes the *dimension* of the kind k , M associates a family of *members* $(M(k)_i)_{i \in [size(k)]}$ with each kind $k \in K$ with $M(k)_i \in RU(k)$ for all $i \in [size(k)]$, *coop* is a control condition over kind names called *cooperation condition*, and *goal* is a graph class expression specifying the *goal*.³

A swarm may be represented schematically as in Figure 4.5 where $s_i = size(k_i)$ and $M_i = M(k_i)$ for $i \in [n]$.

```

name
  initial:    $I$ 
  kinds :     $k_1, \dots, k_n$ 
  size :     $s_1, \dots, s_n$ 
  members:  $M_1, \dots, M_n$ 
  coop:      $c$ 
  goal:      $g$ 

```

Figure 4.5 The schematic representation of a swarm

Definition 23 (swarm computation)

A *swarm computation* is a derivation

$$G_0 \xRightarrow{p_1} G_1 \xRightarrow{p_2} \dots \xRightarrow{p_q} G_q$$

such that $G_0 \in SEM(in)$, $p_j = \sum_{k \in K} \sum_{i \in [size(k)]} r_{jki}$ with a rule r_{jki} of $M(k)_i$ for each $j \in [q]$, $k \in K$ and $i \in [size(k)]$, and *coop* and the control conditions of all members are satisfied. The computation is *successful* if $G_q \in SEM(goal)$

It is a strong requirement that all members must provide a rule to a computational step, because graph transformation rules may not be applicable. In particular, if no rule of a swarm member is applicable to some environment, no further computational step would be possible and the inability of a single member would stop the whole swarm. To avoid this global effect of a local situation, it is assumed that each member has the empty rule $(\emptyset, \emptyset, \emptyset)$ in addition to its other rules. The empty rule is given the lowest priority. In this way, each member can always act and is no longer able to terminate the computation of the swarm. In this context, the empty rule is called *sleeping rule*. It can always be applied, is always parallel independent with each other rule application, but does not produce any effect. Hence, there is no difference between the application of the empty rule and no application even within a parallel step.

As mentioned above, cooperation conditions corresponds to control conditions as introduced in Chapter 3, but the identifiers are kinds instead of rules. In order to use the same typical control conditions as introduced for rules, some considerations should

³For $s = (n_1, \dots, n_l) \in \mathbb{N}_{>0}^l$ and some $l \geq 1$, $[s] = \{(i_1, \dots, i_l) \mid i_j \in [n_j], j \in [l]\}$.

be precisely defined. Let K be a set of kinds in a swarm S . A kind $k \in K$ is said to be *active* in a computation step if each member of kind k applies a rule (including the sleeping rule) specified by its control condition and the requirement of the parallelism independence. The resulting parallel rule is $p(k) = \sum_{i \in [s(k)]} r_{ki}$ with r_{ki} is a rule of member $M(k)_i$. Regular expressions as introduced before yield for kinds when kind activeness is used instead of rule application. More precisely, A derivation obeys a cooperation condition having the form of a regular expression e if the application sequence of the derivation belongs to the language of e denoted $L(e)$ with $SEM(e) = \{G = G_0 \xRightarrow{p(k_1)} G_1 \cdots \xRightarrow{p(k_n)} G_n = G' \mid k_1 \cdots k_n \in L(e)\}$. The expression $k!$ requires that the kind k is active as *long as possible*. A kind can not be active if all members of kind k can apply only the sleep rule. Finally, the cooperation condition *free* requires that all kinds are active in each computation step. This corresponds to the default case introduced in Definition 23.

To enhance the feasibility of the swarm concept, unbounded sizes are also allowed, denoted by \mathbb{N} or \mathbb{Z} . In this case, the only computations allowed are those where in each step all but a finite number of rules are empty. An example of a swarm with unbounded size is the swarm version of a cellular automaton in Chapter 5.

4.3 Examples

This section illustrates the notion of graph-transformational swarms by means of two examples: a very simple ant colony and a swarm that computes Hamiltonian cycles.

4.3.1 Ant colony

A simple ant colony is modeled as the graph-transformational swarm in Figure 4.6. The colony consists of some ants, all of the same kind. They act in directed graphs with some *nest*- and *food*-loops. Initially, an environment graph is simple and – ignoring the loops – unlabeled. All ants wake up simultaneously by accessing edges that exit nodes with *nest*-loops. Then they forage for food by walking through the graph passing one edge per step while each ant marks its individual path. If an ant reaches a node with a *food*-loop, it starts to return to its nest retracing along its own marked path while marking this path with the label *food*. This ant colony is successful whenever a path from a *food*-node to a *nest*-node is created such that all edges of the path are labeled with *food*. Such a path may be called *food*-path. It can be shown that a shortest *food*-path is found by this ant colony with high probability in a linear number of steps provided that the colony is large enough.

The members are obtained by relabeling A and a by A_i and a_i respectively while all other labels are kept. The control condition requires that an ant wakes up at the beginning, but never again. Moreover, the application of the *return*-rule has higher priority than the *found*-rule, which in turn has higher priority than the *forage*-rule. Hence, the ant stops foraging whenever it finds food. From there on, it returns to the nest on the path labeled with a because the return rule has the highest priority. For

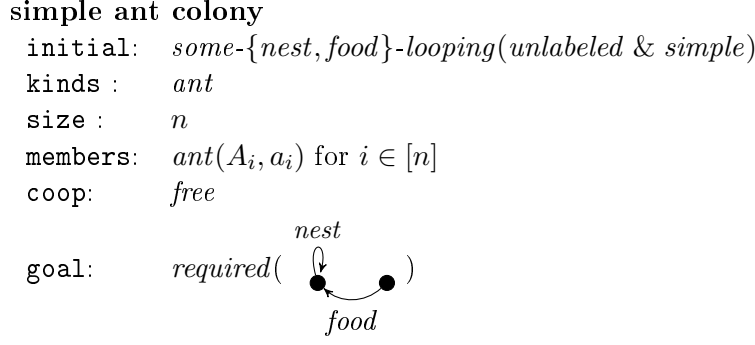


Figure 4.6 The swarm *simple ant colony*

the computations of the ant colony this means for the computations of the ant colony that all ants wake up simultaneously in the first step. The respective rule applications are parallel independent because nothing is removed. If other rules are applied in later steps, then only edges with individual labels A_i and a_i are removed so that each two ants can act in parallel as their rule applications are parallel independent.

A first version of *simple ant colony* has been implemented in the graph transformation tool GrGen.NET. The resulting computation steps of an experiment with a swarm of 7 ants A_1, \dots, A_7 are displayed in rfigure4.8 where the labels a_i with $i \in [7]$ are omitted to allow a good visualization. The initial graph has 26 nodes including a node with a *nest*-loop and two nodes with *food*-loops. In the first step, all ants wake up leaving the *nest* in random directions applying the rule $p_1 = \sum_{i \in [7]} \text{wakeup}_i$ where wakeup_i is obtained from *wakeup* via relabeling A by A_i . In the next two steps applying the rules $p_2 = \sum_{i \in [7]} \text{forage}_i$ and $p_3 = \text{sleep}_3 + \sum_{i \in [7] \setminus \{3\}} \text{forage}_i$, all ants forage for food where ant A_7 succeeds and ant A_3 falls asleep because it gets stuck. (Here again, forage_i is obtained from *forage* by applying the above mentioned relabeling.) A_7 applies its *found*-rule in step 4 and returns to the *nest* in steps 5 and 6 while all other ants forage further or fall asleep because of reaching a dead end.

Please note that an ant can also return before having reached any food, because there may be a *food*-edge inserted by another ant. But this does no harm, because no ant can start to return before at least one ant has found food, and inserted *food*-edges extend paths of *food*-edges originating at *food*-loops.

In contrast, in this way the cooperation of several ants may even lead to a shortest *food*-path, although none of the ants has found one. Consider for example the graph in Figure 4.7. Let A_1 and A_2 be two ants, such that A_1 moves along the path 1, 2, 3, 4 in the first four steps, while A_2 moves along the path 5, 6, 7, 8. Then A_1 has to apply the *found*-rule and A_2 goes to edge 9 with its *forage*-rule. Afterwards, A_1 has to return to edge 3 and A_2 may go to edge 10 with the *forage*-rule. Now, for $i = 1, 2$ there is an A_i -edge as well as a *food*-edge in parallel to edge 10 and hence, A_1 returns to the nest

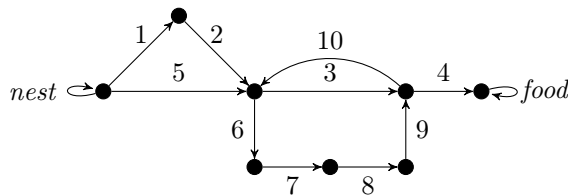


Figure 4.7 A special initial environment for *simple ant colony*

via edge 5 in a single step, whereas A_2 needs two steps although A_2 has found the *food*. Hence, the interaction of both ants delivers a shortest *food*-path.

To see what this ant colony can achieve in general, let p be a simple shortest path from a *nest*-node to a *food*-node in some finite initial environment graph, and let q be its length. Consider a computation of the swarm with at least $2q$ steps. Then each ant runs along a path of length q in the first q steps provided that it does not end up earlier in a node without outgoing edges. As the number of paths of length q (or shorter) is finite, the path p is among the traveled paths with some probability that grows with the size of the swarm. In more detail the following holds: Let D be the maximal outdegree⁴ in the environment graph, then the probability that one ant finds food in q steps is $\geq \frac{1}{D^q}$. Let π_i be the lower bound of the probability that i ants find food in q steps. Then an $(i+1)$ -th ant increases this value by a $\frac{1}{D^q}$ -th fraction of the cases where the i other ants fail, i.e. $\pi_{i+1} = \pi_i + \frac{1}{D^q}(1 - \pi_i)$. By induction, one gets $\pi_i = 1 - (1 - \frac{1}{D^q})^i$ which proves that the probability of success is nearly 1 if many ants are involved. In other words, the ants constitute a Bernoulli process with the ratio $1 - \frac{1}{D^q}$. Hence, there is a good chance that some ant can apply the *found*-rule in step $q+1$. This ant retravels p and marks it as a *food*-path in the next $q-1$ steps such that a shortest path is established in $2q$ steps with high probability, i.e., in a number of steps linear in the number of nodes of the initial environment as q is always smaller than the number of nodes.

4.3.2 Hamiltonian cycles

Consider an unlabeled and simple graph where each node gets an extra loop with a label in $\Delta \subseteq \Sigma$. A Δ -cycle is a cycle that visits exactly one node with a δ -loop for each $\delta \in \Delta$ and no other node. If Δ is the set of nodes and each node v has a v -loop, then a Δ -cycle is a Hamiltonian cycle. Consequently, the search for a Δ -cycle is a generalization of the Hamiltonian cycle problem and an *NP*-complete problem itself.

The graph-transformational swarm in Figure 4.9 searches for Δ -cycles using two rules. The swarm has two kinds given by the rules *nodecopy* and *edgcopy*. An application of *nodecopy* generates a copy of a node with a v -loop and decorates it with a b - and an e -loop. An application of *edgcopy* copies an edge of the original graph consuming an e -loop at a copy of the source node and a b -loop at a copy of the target node. All members are identical to their kinds. Each two applications of *nodecopy*-rules are par-

⁴The outdegree of a node is the number of its outgoing edges

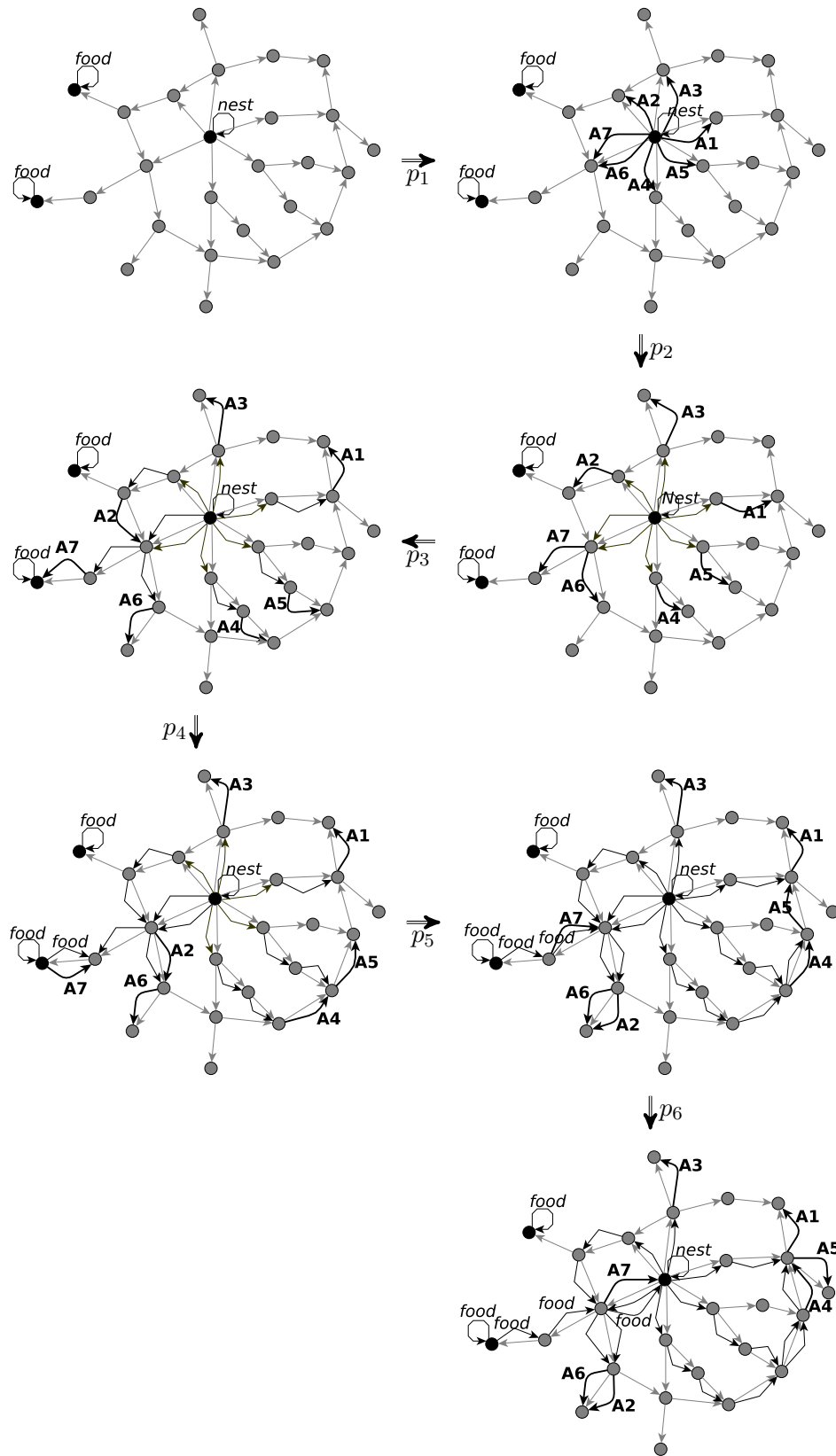


Figure 4.8 A sample computation of the *simple ant colony swarm*

Δ -cycle

initial: *all- Δ -looping(unlabeled & simple)*
kinds : *gtu(nodecopy), gtu(edgecopy)*
size : n_1, n_2
members: $m_{1i} = \text{gtu}(\text{nodecopy})$ for $i \in [n_1]$
 $m_{2j} = \text{gtu}(\text{edgecopy})$ for $j \in [n_2]$
coop: *nodecopy ; edgecopy*
goal: *required(Δ -cycle copy)*
rules:

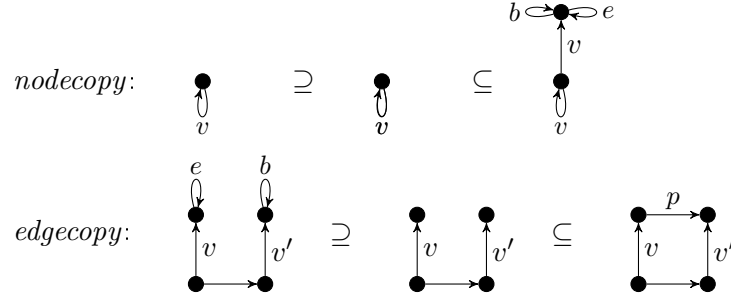


Figure 4.9 The swarm Δ -cycle based on the rules *nodecopy* and *edgecopy*

allel independent as they do not remove anything. Therefore, all n_1 *nodecopy*-members can produce n_1 node copies in parallel. The *edgecopy*-members must sleep in the first step as they need node copies to become active. Due to the cooperation condition, the *nodecopy*-members are requested to sleep in the second step. Two applications of *edgecopy*-rules are parallel independent if they consume different b - and e -loops. Hence, there are up to n_1 edge copies after the second step. Because the target of an edge copy may be the source of another edge copy, the edge copies form paths and cycles corresponding to paths and cycles in the original graph. As paths and cycles may be copied several times, the *edgecopy*-step yields a multiset of paths and cycles of the initial graph. If one is lucky, then there is a cycle copy among the created paths and cycles that corresponds to a Δ -cycle. This is the goal. Which is reached with high probability if the sizes of the swarm kinds are large. This can be proved similar to argumentation in 4.1 as follows. If $n_1 \geq k \cdot n$ where n is the number of nodes of the considered initial graph and $k \in \mathbb{N}$, then there is a good chance that all nodes have got about k copies after the *nodecopy*-step. Hence, each edge can be copied up to about k times in the *edgecopy*-step. If d edge copies are made where d is the number of elements of Δ , the resulting set of path and cycle copies represent a multiset of paths and cycles of the original graph the sum of lengths of which is d . Let N be the number of all such multisets and M be the number of multisets that correspond to Δ -cycles. Note that the resulting set of path and cycle copies may contain a single element that corresponds directly to a Δ -cycle or it may contain several path copies that compose a Δ -cycle. Then $\frac{M}{N}$ is the probability that d edge copies find a Δ -cycle. Let now π_i be the probability that $i \cdot d$ edge copies find a Δ -cycle, and add further d edge copies.

Then $\pi_{i+1} \geq \pi_i + \frac{M}{N}(1 - \pi_i)$ because the $(i + 1)$ -th set of edge copies may be successful where the others failed and, in addition, the paths found by the $(i + 1)$ -th set may compose with the paths found by the other edge copies to Δ -cycles. As in 4.1, one gets by induction $\pi_i \geq 1 - (\frac{N-M}{N})^i$ for $i \in \mathbb{N}$ such that the sequence π_i converges toward 1 with growing i .

This swarm solves an *NP*-complete problem (that covers the Hamiltonian-cycle problem) in two steps with high probability if the sizes are properly chosen. The task to check whether there is a Δ -cycle copy is not performed by the swarm. But it could be extended in such a way that this test needs d steps.

4.4 Stochastic control

The control conditions considered up to now regulate the application sequence of rules. This type of control is also known as *Language type conditions* (for more details see for example [62]). This section introduces another type of control condition, a control condition that regulates the matching process based on probability computations. It will be called *stochastic control*.⁵

A stochastic control is a control condition that is associated with a rule r . It controls the non-determinism of choosing a match of r in a given graph G . It requires that a match is chosen with a probability proportional to an associated value. Formally, this can be done in the following way:

Definition 24 (stochastic control)

Given a rule r and a graph G . Let $M_G(r) = \{m_1, \dots, m_k\}$ be the set of all feasible⁶ matches of r in G , let $f : M_G(r) \rightarrow \mathbb{R}_{>0}$ be a mapping that associates to every match m_i a value $f(m_i) = f_i$ for $i = 1, \dots, k$. A stochastic control condition denoted $\wr f$ requires that a match m_j for $j \in \{1, \dots, k\}$ is chosen with the probability $\frac{f_j}{\sum_{i=1}^k f_i}$.

f_i can correspond to a label of a given edge in m_i . It can also be the result of a parametrized function of one or more labels in m_i . In literature, one encounters several examples of distribution functions that control the behavior of stochastic processes based on some parameters. The parameters make it possible to produce a choice behavior situated between two extremities: (1) the random choice where all possible matches have the same probability and (2) the so called *greedy* choice where a match with the highest value is selected.

Let $v \in \mathbb{R}_{>0}$ be a value associated with a match $m \in M_G(r)$. Let us consider two examples of such parametrized functions:

- The first example is the function $f(v) = (v + \beta)^\alpha$. This corresponds to the function introduced in Chapter 2. It has been used by biologists to model the selection

⁵Note that in the literature there exists the notion of stochastic graph transformation systems[41]. It refers however to a stochastic choice of rules and not of matches.

⁶A feasible match is a match for which the underlying morphism satisfies the dangling conditions.

behavior of real ants when they forage for food and when they have to choose between two directions. The parameter β controls the delay needed for further exploration of the environment at the beginning of the computation process. The parameter α determines the importance of the difference of values in the decision process.

- The second function is $f(v) = e^{\frac{v}{T}}$ which is a function often used in the field of reinforcement learning [94]. The parameter T is a parameter called temperature. The underlying distribution is called *Boltzmann or Gibbs* distribution. The higher the temperature is, the more the choice behavior tends to be random. Low temperatures cause a greater difference in probability for matches that differ in their values, tending to choose greedily with temperatures close to zero.

Stochastic control condition can be implemented based on the "roulette wheel selection" also known as "Fitness proportionate selection" in the field of genetic algorithms. The idea is to assign to each of the possible matches an interval with a length equal to the corresponding probability (surface in a wheel proportional to its probability). Afterward, a random number is generated such it belongs to one of the intervals (turning the wheel). Section 4.5 proposes a concrete implementation by means of graph transformation units. But first a pheromone-driven ant colony that uses stochastic control illustrates how to use the control in the graph transformation framework.

4.4.1 Pheromone-driven ant colony

The example of the simple ant colony is extended here to illustrate stochastic control. The pheromone-driven ant colony is a swarm that models an ant colony, the ants of which forage for food by means of a simple pheromone mechanism. The sample graph-transformational swarm is presented in Figure 4.10.

The swarm consists of some ants, all of the same kind. They act in directed graphs with a *nest*-loop and some *food*-loops. The node with the *nest*-loop has some further unlabeled loops that represent the actual food stock. All other initial edges are labeled by a positive integer representing a pheromone rate. *nest-food*-connectedness is assumed, meaning that the paths from the *nest*-looped node to some *food*-looped node visit all nodes. Moreover, it is assumed that the underlying environment graph is simple, meaning that there are no parallel pheromone-labeled edges. This class of graphs is denoted by *(nest & food*)-looping(simple & pheromone-labeled & nest-food-connected)*. During swarm computations, further edges appear and disappear.

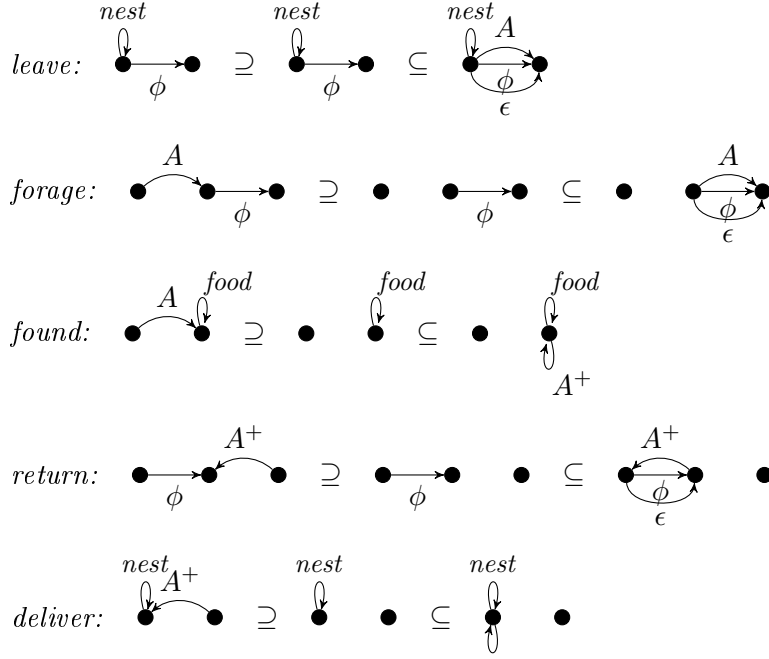
The kind *ant* defines the potential activities of an ant by means of five rules and some priorities. It can *leave* the nest by placing an *A*-edge and an ϵ -labeled edge in parallel to a pheromone-labeled edge with the *nest*-looped node as source. Then it can *forage* for food by walking through the graph passing one pheromone-labeled edge per step and placing a parallel ϵ -edge. The label *A* refers to the ant, and ϵ is an integer to be added to the pheromone value. If an ant reaches a *food*-node, then the rule *found* is applied changing the label *A* into A^+ and indicating in this

pheromone-driven ant colony

initial: $(\text{nest} \ \& \ \text{food}^*)\text{-looping}(\text{simple} \ \& \ \text{pheromone-labeled} \ \& \ \text{nest-food-connected})$
 kinds : $\text{ant}, \text{update}$
 size : $n, 1$
 members: $\text{ant}(A_i, A_i^+)$ for $i \in [n]$, update
 coop: $(\text{ant} ; \text{update})^*$
 goal: $\text{required}(\text{stock} \geq b)$

ant

rules:



control: $\setminus \text{leave} \setminus_{\phi^\alpha} < \setminus \text{forage} \setminus_{\phi^\alpha} \mid \text{found} \mid \setminus \text{return} \setminus_{\phi^\alpha} \mid \text{deliver}$
 $\setminus \text{forage} \setminus_{\phi^\alpha} < \text{found}$
 $\setminus \text{return} \setminus_{\phi^\alpha} < \text{deliver}$

update

rules:



control: $\parallel \text{update} \parallel !$

Figure 4.10 The swarm *pheromone-driven ant colony* with the kinds *ant* and *update*

way that the ant takes food. In this state, it moves back using the rule *return* until it can *deliver*, which adds a food unit to the stock. Note that the returning ants pass edges from target to source so that the same paths are used as for foraging. Moreover, an ant leaves the amount ϵ of pheromone along the return paths too. The pheromone values of the passed edges are not updated immediately, but in the next computational step. This allows several ants to pass the same edge in the same step.

The control condition requests some priorities and stochastic control. An ant can only leave the nest if it cannot do anything else, i.e., if neither the label A nor A^+ is around. In other words, it leaves the nest at the beginning and after each delivery. Moreover, foraging for food stops whenever food is found and moving back stops whenever the nest is reached. Further control is provided by the labels A^+ and A . As long as A is present, only the rules *forage* and *found* may be applied. As long as A^+ is present, only *return* and *deliver* may be applicable. The application of *found* turns a foraging phase into a returning phase that ends with *deliver*. The rules *leave*, *forage* and *return* are applied following a stochastic control requiring that the number of ants that pass an edge corresponds to the pheromone value of the edge. More precisely, let l be an ant that can pass the edges e_1, \dots, e_k with pheromone values ϕ_1, \dots, ϕ_k in the next step, then e_j is used with the probability $\frac{\phi_j^\alpha}{\sum_{i=1}^k \phi_i^\alpha}$ where the parameter α can be chosen in a suitable way. The larger α is, the more the effect of the pheromone values is intensified in the heuristic choice.

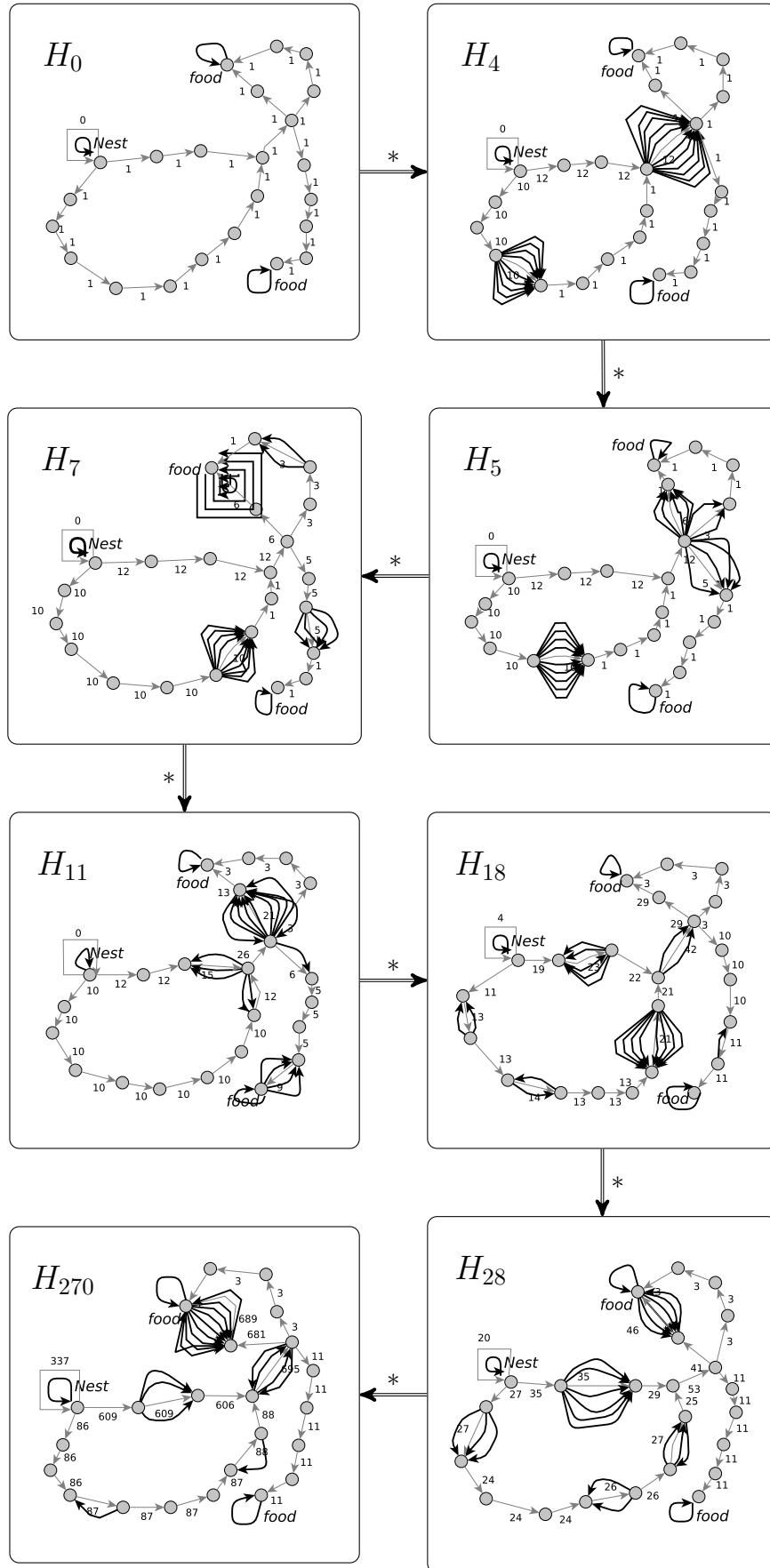
Due to the *nest-food*-connectivity of the environmental graph, an ant can always act. If the A -edge points to a *food*-looped node, then the rule *found* can and must be applied. Otherwise the A -edge has a target with another outgoing edge so that *forage* can be applied. If there is an A^+ -loop, then *return* can be applied. To match the left-hand side of the rule in this case, its A^+ -edge must be mapped to the A^+ -loop. This is possible because matches are not assumed to be isomorphic images. If there is an A^+ -edge pointing to the *nest*-looped node, then *deliver* can and must be applied. Otherwise, the A^+ -edge points to a node with an incoming edge so that *return* can be applied. If all other rules fail, *leave* is allowed and possible.

The members of kind *ant* are obtained by relabeling A and A^+ by A_i and A_i^+ resp. for $i = 1, \dots, n$ where n is the chosen size of the ant colony. All other labels are kept as they are.

As all rule applications remove only edges with labels A_i and A_i^+ , all rule applications are pairwise parallel independent if they concern different labels. In other words, the maximal parallel computation steps can be performed whenever an applicable rule is chosen for each ant.

The cooperation condition requires that after each action of the ants an *update* of the pheromone values takes place. The only member equals the kind and provides a single rule that adds ϵ to each pheromone-labeled edge for each parallel ϵ -labeled edge.

The control condition requires that the *update*-rule is applied with maximal parallelism

Figure 4.11 A sample computation of the *pheromone-driven ant colony swarm*

as long as possible. The applications of the *update*-rules are parallel independent if they update different pheromone-labeled edges. Therefore, *update* needs m steps where m is the maximum number of parallel ϵ -edges.

Finally, the goal specifies graphs where the stock, i.e. the number of extra loops at the *nest*-looped node, exceeds a given bound b that can be chosen freely.

From the description of this swarm, it is clear what the computations look like. The ants act in parallel, each applying one of its five rules according to applicability and priority. In the first step, all ants leave the nest. Later in the computations, all five types of rules may occur simultaneously. After each ants action step, an update takes place. The alternation between ant action and update can go on for ever, but can be stopped if the stock is large enough. Will this event occur eventually? It is assumed that the initial graphs are *nest-food*-connected so that there are paths from the nest to each *food*-labeled node in particular. The ants use those paths with some probability depending on the pheromone values. Consequently, the ants come back to the nest after they found food with some probability so that the stock increases with some probability if the computation runs long enough and the number of ants is large enough. This can be guaranteed by additionally assuming that the initial environments are finite and cycle-free because then every ant finds food and returns to the nest eventually. The pheromone mechanism favors short paths over long ones. The fastest way to increase the stock is by running a shortest path from *nest* to *food* and back. Short paths get some extra pheromone earlier than long ones so that they will be used in the further computation with even higher probability. This reasoning shows that there is a correlation between the length of paths and the number of computation steps needed to fill the stock.

Because this is a very first example of graph-transformational swarms, it has been kept simple. In particular, the kind *update* could be designed in a more sophisticated way by adding evaporation rules. Moreover, the only member *update* could be replaced by *update*-members that are related to the pheromone-labeled edges so that the pheromone updating is also in the style of swarms.

The *pheromone-driven ant colony* swarm has also been implemented in the graph transformation tool GrGen.NET. An experimental computation with a swarm of 20 ants is documented in Figure 4.11. For better visualization, the labels of the ants have been omitted and the loops representing the food stock replaced by a single loop labeled with the number of food units. The initial graph H_0 has 23 nodes including a node with a *nest*-loop and two nodes with *food*-loops. The initial pheromone values of all edges correspond to $\phi = 1$. In the probability function, we use $\alpha = 2$. The seven further displayed graphs H_i for $i \in \{4, 5, 7, 11, 18, 28, 270\}$ are the graphs after the i -th step of the ants and the following update each. The graph H_4 represents the resulting graph after four *ant*-steps. More precisely, in the first *ant* step all ants leave the nest, however the swarm is split into two groups of almost the same size 9 and 11. This is due to the *pheromone-driven* action of ants and the equal initial pheromone values. Afterwards all ants apply their *forage*-rules three times. The edges visited by each group can be easily recognized in H_4 , since their initial values are augmented by the underlying group's

number of members. The graph H_5 shows the result after the 5th *ant*-step. One can see how all members go forward applying their forage rules again. However the group of 11 members splits into three subgroups when arriving at the node, say u , with three outgoing edges. In the 7th *ant*-step which generates H_7 , a group of 5 ants find the *food*-node, say f_1 , while all other ants forage further. In the 11th *ant*-step, 11 ants have found food and are returning to the nest. The other members still forage. H_{18} displays the results of the 18th *ant*-step. The first ants have delivered 4 units of food, in addition one can see that the path between u and f_1 is slowly starting to be preferred. In H_{28} the ants already have already 28 steps, and 20 units of food are delivered. The path between u and f_1 is frequently walked through meanwhile. H_{270} displays the graph after 270 *ant*-steps with 337 food units. Based on the pheromone values, one can see that ants prefer the shortest path between the *nest*- and one of the *food*-nodes. The computation may be terminated whenever the chosen bound of the food stock is reached.

4.5 Modeling: practical considerations

One advantage in the field of graph transformation is the existence of various tools dedicated to model graph transformation systems. After the introduction in Chapter 3 that compared different tools regarding the requirements in this thesis and presented a first modeling of parallelism in the selected tool GrGen.Net, the present section provides a discussion about the implementation of graph transformational swarms using first the notion of stepwise control which is tool independent. Afterwards, an explicit coding in the tool GrGen.net is provided.

4.5.1 Stepwise control

Stepwise control is a general method for modeling and analyzing control conditions that are based on regular expressions. For explicit use below, we sketch a special class of the stepwise control (conditions) presented in [35]. This special class consists of finite state automata over rules and negated rules with an arbitrary number of initial states. More concretely, such a control condition is a system $C = (S, J, F, \Delta)$ where S is a finite set of *control states*, $J \subseteq S$ is a set of *initial control states*, $F \subseteq S$ is a set of *final control states* and $\Delta \subseteq S \times (P \cup \{\neg r \mid r \in P\} \cup \{\lambda, \neg\lambda\}) \times S$ is the *transition relation*. As usual, C can be represented as a graph with S as nodes and Δ as edges. The $\neg r$ -edges are called *negated edges*. A (possibly empty) path with only negated edges and λ -edges is a *negated path*. It is a *negated path for a graph G* , if no rule on the path is applicable to G . A derivation $G_0 \xRightarrow{r_1} \dots \xRightarrow{r_n} G_n$ is specified by C , if there is a sequence s_0, \dots, s_n of control states such that $s_0 \in J$, there is a negated path for G_n from s_n to some final state, and for $i = 1, \dots, n$ there is an r_i -path for G_{i-1} from s_{i-1} to s_i , i.e., a path from s_{i-1} to s_i that consists of a negated path for G_{i-1} followed by an r_i -edge.

Figure 4.12 visualizes the state transitions of the above introduced control condition (*wakeup* ; (*forage* | *found* | *return*)*) & (*return* > *found* > *forage*). The nodes represent control states and the edges visualize the transition relation. An edge labeled r means

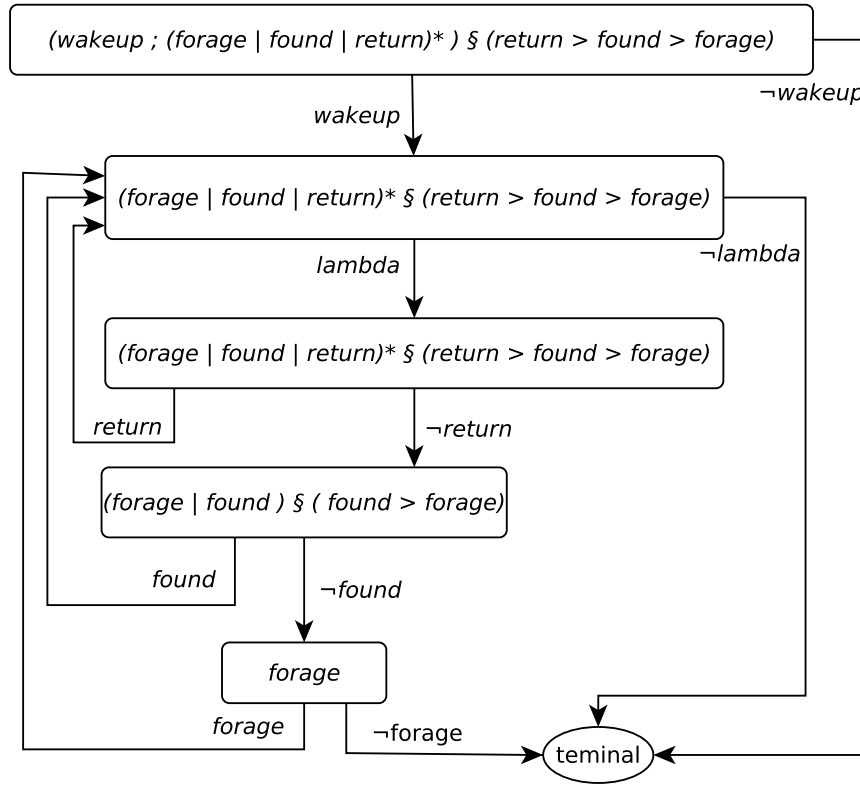


Figure 4.12 State transition diagram of the control condition $(wakeup ; (forage | found | return)^* \& (return > found > forage))$

that the rule r is applicable to the current graph and if labeled $\neg r$ (negated rule) it means that r is not applicable. *lambda* and $\neg lambda$ stand for the arbitrary choice between a repeat of the computation or not. In this visualization, it is easy to see that for example after the *wakeup* step a direct derivation using the rule *forage* corresponds to a step given by a negated path $\neg return \neg found$ followed by a *forage*-edge.

Further examples of stepwise controls including as-long-as-possible expression can be found in [35] and [67]. In [67], there is also a description of how to compose a stepwise control from different ones.

Swarm computations by means of stepwise controls

Let $S = (in, K, size, M, coop, goal)$ be a swarm and let us assume that each member $M(k)_i$ has a stepwise control $C_{ki} = (S_{ki}, J_{ki}, F_{ki}, \Delta_{ki})$ for $k \in K$ and $i \in [size(k)]$. A swarm computation $G_0 \xRightarrow{p_1} G_1 \xRightarrow{p_2} \dots \xRightarrow{p_q} G_q$ such that $G_0 \in SEM(in)$, $p_j =$

$\sum_{k \in K_j} \sum_{i \in [s(k)]} r_{jki}$ with $K_j \subseteq K$ is the set of active kinds ⁷ in step j , and r_{jki} is the chosen rule from the stepwise control C_{ki} to participate in the computation step j . r_{jki} can be the *sleeping* rule if no other rule is applicable. Another important condition of applicability is that each two rule applications in one computation step are parallel independent.

A computation $G_0 \xRightarrow[p_1]{p_1} G_1 \xRightarrow[p_2]{p_2} \cdots \xRightarrow[p_q]{p_q} G_q$ can be performed as follows: At the beginning, the state of all stepwise controls of all members is an initial one. Let us denote such a state by s_{0ki} for C_{ki} and $i \in [size(k)]$ and $k \in K$ with $s_{0ki} \in J_{ki}$. In step j for $j \in [q]$, first the cooperation condition specifies the set K_j of active kinds. Afterwards, each member $M(k)_i$ ($i \in [size(K_j)]$) of an active kind $k \in K_j$ chooses an applicable rule r_{jki} using its stepwise control C_{ki} with respect to the parallelization condition. After that, the parallel rule $p_j = \sum_{k \in K_j} \sum_{i \in [s(k)]} r_{jki}$ is applied. The states of the stepwise controls of the members that have participated in the computation step j with a rule, different to the *sleeping* rule, are adapted accordingly. For all other members, the state of their stepwise control is kept unchanged.

Therefore, given stepwise controls for the members in a swarm, the computation of the swarm can be performed based on the cooperation condition which designates the members that can perform a local computation step. All chosen rules are applied and the states of the participated stepwise controls are adapted accordingly. An important ingredient in this parallelization step is the condition that each two rule applications are parallel independents. How to manage such a verification is generally complex. More precisely, assuming that n members are active in a computation step, there are $\binom{n}{2}$ (binomial coefficient indexed by n and 2) verification that should be performed. However, in practice, there are some strategies to reduce the number of eventual conflicts between two different members. The most simple form is to design kinds such that if two rules are from two different members, then their applications are always parallel independent. This is the case in the simple ant colony example. Another possibility is to consider special cases of members that are assigned to special regions in graphs and are responsible to manage the resources (edges or nodes that can be deleted) in this region. The next two chapters introduce swarms that use this technique.

4.5.2 Modeling in the tool GrGen.Net

This subsection discusses how to implement graph-transformational swarms in the tool GrGen.Net using the example of the simple ant colony as introduced above in this chapter. The discussion starts by the implementation of the elementary component in the framework, namely the rules.

Rules

Rules can be easily implemented using the syntax of GrGen.Net. The Listing 4.1 displays an example of the four rules *Wakeup*, *Forage*, *Found* and *Return*. They correspond to

⁷In this notation, a special case of *coop* (but the only one used in this thesis) is considered. *coop* specifies in every computation step j a set of active kinds $K_j \subseteq K$.

the rules with the same name (up to capitalizing) in the kind *ant*. The difference here is that the rules are given an edge *A* of type *Ant* as argument. The reason for that will be explained below in detail. The code can be interpreted as follows: Every rule consists of three parts. The first line declares the name of the rule and the arguments that it takes as input. The second part consists of the lines before the keyword *modify* (lines 2, 3 and 4 in the rule *Wakeup*). It specifies the left hand side of the rule. The third part in the *modify* block specifies what the rule should modify, specifying implicitly the gluing graph and the right hand side of the underlying rule. Let us describe the rule *Forage* starting in Line 11. Line 12 contains the graph consisting of three nodes *x*, *y* and *z* such that *y* is an outgoing neighbor of *x* and *z* is an outgoing neighbor of *y*. Line 13 extends the graph by an edge *A* (of type *Ant*) from *x* to *y*. The Line 15 requires that an ant path *a* is added from *x* to *y*. In the block starting with the keyword *eval*, all computations including assignments of label values and attributes are performed. In this case, the name and the index of the created ant path are changed accordingly. After that, a copy of the edge *A* is created from *y* to *z* in Line 20. In Line 21, the old edge *A* is deleted.

Listing 4.1 The rules of the unit ant

```

1: rule Wakeup (-A:Ant->){
2:  x:Node--:Nest->x ;
3:  x- A->x ;
4:  x-->y:Node ;
5:  modify{
6:    x-newA:copy<A>->y;
7:    delete(A);
8:  }
9: }
10:
11: rule Forage (-A:Ant->){
12:  x:Node-->y:Node-->z:Node;
13:  x -A-> y;
14:  modify{
15:    x-a:AntPath->y;
16:    eval{
17:      a.name= "a"+A.indice;
18:      a.indice = A.indice;
19:    }
20:    y-newA:copy<A>->z;
21:    delete(A);
22:  }
23: }
24:
25: rule Found (-A:Ant->){
26:  x:Node-A->y:Node--:Food-> y;
27:  modify{
28:    y-:FoodPath->x;
29:    y-newA:copy<A>->x;
30:    delete(A);
31:  }
32: }
33:
34: rule Return (-A:Ant->){
35:  x:Node-->y:Node-->z:Node;
36:  x -a:AntPath->y;
37:  z -A-> y;
38:  if { a.indice==A.indice;}
39:  modify{
40:    y- newA:copy<A>->x;
41:    delete(A);
42:    y -f:FoodPath-> x;
43:  }
44: }

```

Kinds and members

In GrGen.Net, it is possible to use a rule as a structuring device. It is possible to call other rules that are to be applied using a syntax based on regular expressions. In the example of ant colony the following

part of code is used to implement the control condition of the kind *ant*:

Listing 4.2 the unit ant

```

1: rule AntUnit(-A:Ant->){
2:   modify{
3:     exec ( Return(A) || Found(A)|| $[Forage(A)] );
4:   }
5: }
```

The operation $r_1||r_2$ in GrGen.Net stands for apply r_1 or r_2 such that r_1 has a higher priority. That is, the expression in Listing 4.2 is semantically equivalent to the expression $(forage|food|return) \& (return > found > forage)$. The expression $\$[Forage(A)]$ requires that the match of the left-hand side graph of $Forage(A)$ is chosen randomly⁸. In order to encode that a rule belongs to a given unit, the present implementation uses the notion of rule arguments. The rules of a given kind are all given the same argument. In the example used, all rules get the ant edge A (the unique kind) as argument. For the specification of members, the notion of classes and attributes is used in GrGen.Net. The edge A is declared as *Ant* which is a subclass of the main edge class *Edge* and has two attributes. All such specifications are declared in a so called graph model file. The file in our example is listed in Listing 4.3. The members are obtained using the attribute *index*. The attribute *name* is used for visualization purposes. Its value during computations is A_{index} .

Listing 4.3 the graph model

```

1: edge class Food;
2: edge class Nest;
3: edge class Ant {
4:   name:string;
5:   index:int;
6: }
7: edge class AntPath {
8:   name:string;
9:   index:int;
10: }
11: edge class FoodPath;
```

The classes *Food*, *Nest* and *Foodpath* are primarily defined in order to permit a good visualization of the computation steps, given a different color for each class, it is more easy to follow the behavior of the ants on the graph.

⁸In GrGen.Net and in order to accelerate the matching process, the first found matches in an internal list are chosen.

Swarm computation

A swarm computation is implemented using the notion of maximal parallel matching. In this regard it is important to note the difference between maximal parallelism and maximal matching which is discussed in Section 3.9. In simple ant colony, each two direct derivations from two different members are parallel independent. For this reason, the maximal parallel matching can be used - without restrictions - to match the different members existing in the swarm. Listing 4.4 corresponds to the code in the GRS "antColony.grs" file. It contains the code that controls the execution of computation. It also contains all the operations needed for creating and dumping graphs with a debug support. In Line 8 and 9, the sequence of the computation sequence is coded. The *InitWorld* stands for a rule that creates an environment graph. *Initmember* is a rule that calls the rule *Wakeup*. *Initmember* is called with maximal matching, meaning that all members leave the nest in parallel. After that, the rule *SwarmComputation* is repeated until the *Goal* is met. The rule *SwarmComputation* has an empty left hand side graph, meaning that it is always applied. It calls the rule *MatchMember* with maximal matching, the left hand side of which contains the ant edge *A*. This applies the rule *AntUnit* introduced above. Together, all ant edges are matched and for each one, the *AntUnit* is called using the corresponding edge as argument.

Listing 4.4 The execution statement in the ".grs" file

```

1: new graph antColony
2: debug set layout Organic
3: dump set node Node shape circle
4: .....
5: .....
6: randomseed 40
7: debug set layout option INITIAL_PLACEMENT AS_IS
8: debug exec (InitWorld ;> [InitMember] ;>
9:             (!Goal&& SwarmComputation)* )

```

Listing 4.5 swarm computation

<pre> 1: rule SwarmComputation{ 2: modify{ 3: exec([MatchMember]); 4: } 5: } 6: rule MatchMember{ 7: -A:Ant->; 8: modify{ 9: exec(AntUnit(A)); 10: } 11: } </pre>	<pre> 12: rule InitMember{ 13: A:Ant; 14: modify{ 15: exec(\$[Wakeup(A)]); 16: } 17: } 18: test Goal{ 19: x:Node -:Nest->x ; 20: :Node -:FoodPath->x; 21: } </pre>
--	---

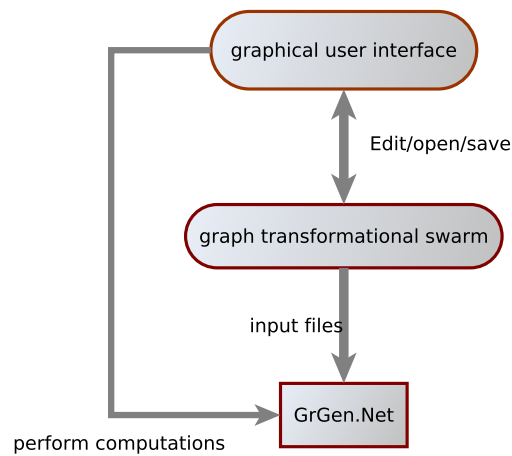


Figure 4.13 A high level schema of the graphical editor and its components

4.5.3 A graphical user interface

Within the scope of this thesis a prototype of a graphical interface that permit a rapid and visual development of swarms is developed. Figure 4.13 illustrates the relationship between the graphical user interface, the graph-transformational swarm framework and the tool GrGen.Net. The elements in the framework can be loaded, edited and saved using the graphical interface. Given a graph transformational swarm, input files for GrGen.Net can be generated either using textual scripts or using the graphical interface. There is also the aim to develop utilities in the interface to control directly the computation process.

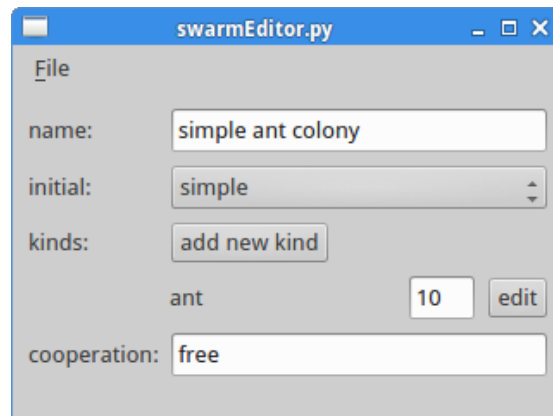


Figure 4.14 The swarm editor

The graphical user interface is composed of several widgets, Figures 4.14, 4.15 and 4.16

are screen-shots of the most important of them. All three images are taken when modeling the *simple ant colony* swarm as introduced in Chapter 4. Figure 4.14 displays the swarm-editor widget which offers to the user the possibility to specify a swarm. It is possible to give the name, the initial graph class expression which can be selected from a predefined set, the kinds and the cooperation condition.

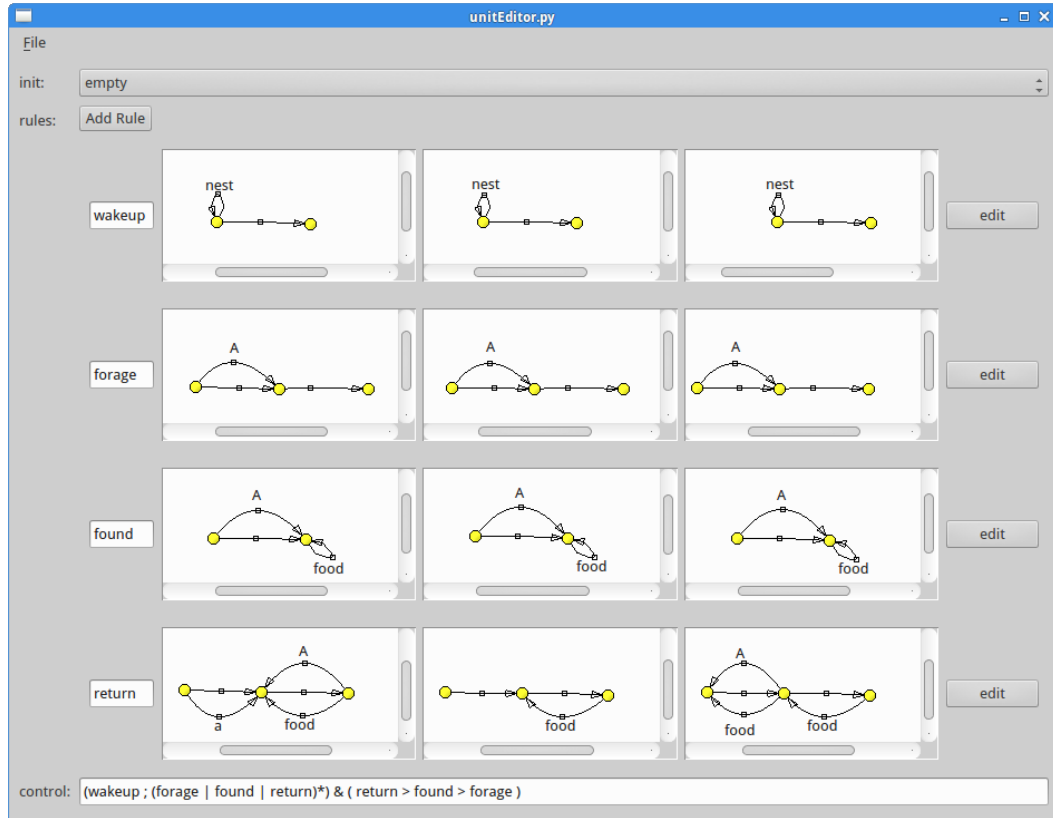


Figure 4.15 The kind editor

The kinds are added in an interactive way by pushing the button "add new kind". As consequence, the *kind-editor* which dedicated to edit the components of a kind, is opened. Figure 4.15 illustrates the edited kind *ant*. The kind-editor allows to specify all components needed for a kind. Here, it is possible to add rules by pushing the button "Add Rule". This is achieved using a third widget called the *rule-editor* as depicted in Figure 4.16.

The *rule-editor* allows a comfortable editing of rules. The left-hand side is created in the left window in the *rule-editor* widget. The created left-hand side graph is automatically copied to the middle and right windows which are reserved for the gluing and right-hand side graphs respectively. Afterwards, modifications can be performed to the right window. If an element is deleted from the right-hand side graph, the gluing graph is accordingly and automatically adapted. The edited components of a swarm can be saved and reedited later. The swarm can also be exported to run in GrGen.Net. The first

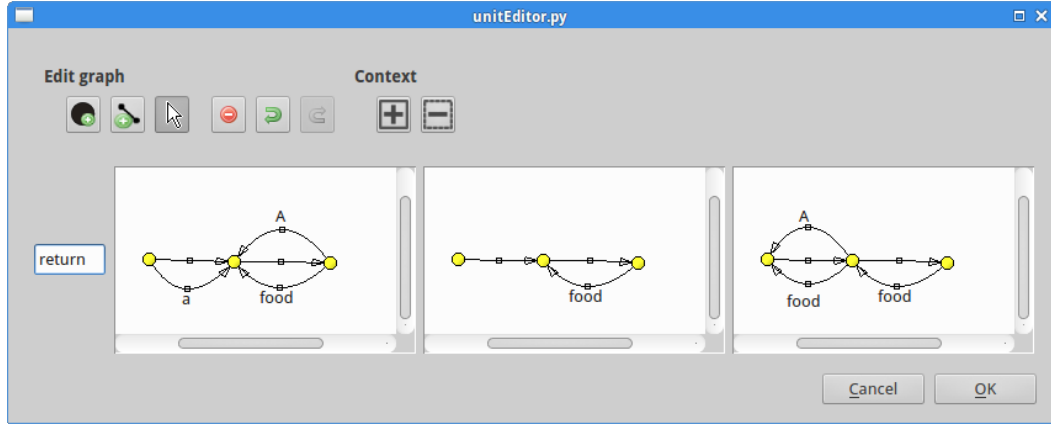


Figure 4.16 The rule editor

tests and experiments using the developed prototype show promising results, however several features should be added.

4.5.4 Stochastic control as a graph transformation unit

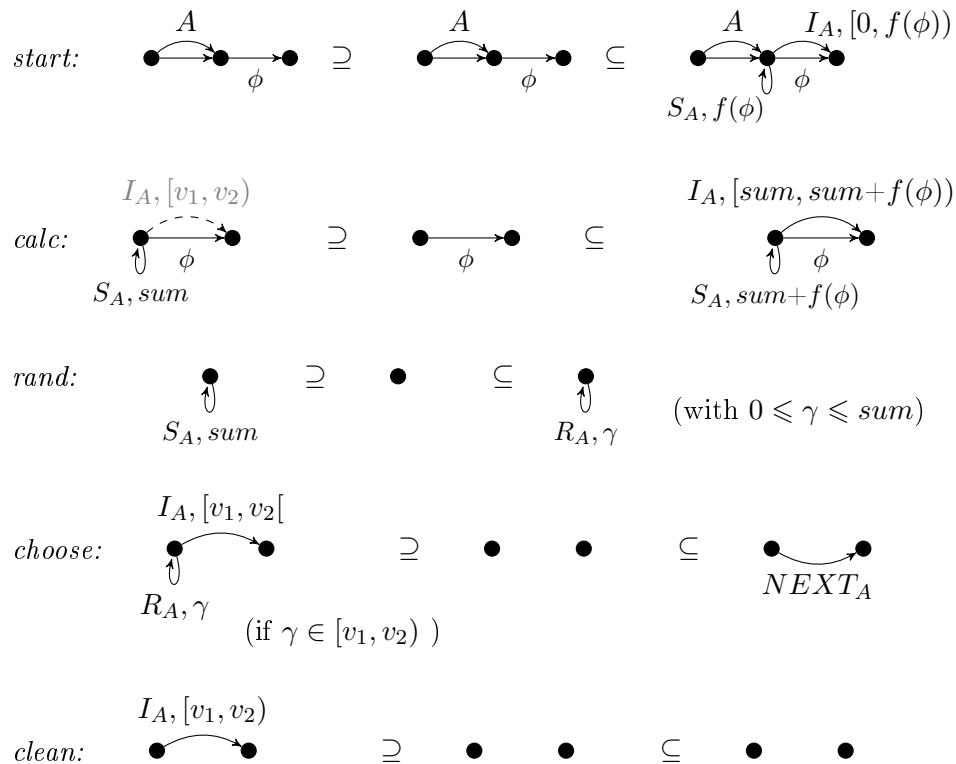
This subsection illustrates how it is possible to model a stochastic control condition using graph transformational units. Stochastic control conditions can be implemented based on the "roulette wheel selection" : One can imagine that each of the matches m_i becomes a surface (bin) proportional to its function value on a roulette wheel. Turning the wheel corresponds to generating a random number in the range $[0, \text{sum}]$ where sum is the summation of the values of all possible matches. To illustrate how such a selection can be performed by a graph transformation unit, let us assume the following concrete situation. An ant A modeled by an A -edge has to choose one edge stochastically from all possible next edges. The choice is based on a distribution function f having as argument the pheromone value for each next edge. Figure 4.17 gives an example of a unit that performs a stochastic calculation.

The rule *start* initiates the calculation. It chooses a next edge with a pheromone ϕ arbitrarily, adds a so-called *sum*-loop in the target node of A and creates a so-called *interval*-edge parallel, and with the same direction, to the chosen next edge. The *sum*-loop is labeled by $S_A, f(\phi)$ and the *interval*-edge by $I_A, [0, f(\phi))$. S_A and I_A are fixed names. They are used to identify the two special edges *sum*-loop and *interval*-edge respectively⁹. The value $f(\phi)$ is used as initial value of the sum in the *sum*-loop as well as the supremum of the right-open interval in the *interval*-edge. The infimum of the interval is 0. The rule *calc* chooses a next edge with a pheromone value ϕ that does not have a parallel *interval*-edge yet and creates a one. The created interval has as infimum the current sum and supremum $\text{sum} + f(\phi)$. The sum in *sum*-edge is changed

⁹The index A is replaced by the index of the underlying member when relabeling, to permit the identification of the considered ant. This is necessary, when more than one ant have the same target node.

next edge stochastic

rules:

control: *start* ; *calc*! ; *rand* ; *choose* ; ||*clean*||**Figure 4.17** A unit that performs a stochastic choice

to be $sum + f(\phi)$. The rule *rand* generates a random number λ between 0 and sum . It replaces the *sum*-loop by a *random*-loop labeled R_A, λ (R_A has the same role as S_A and I_A and identifies the labeled edge as a random edge). The rule *choose* chooses the *interval*-edge with the interval $[v_1, v_2[$ such that $\lambda \in [v_1, v_2)$. It replaces the chosen *interval*-edge by a *next*-edge labeled $NEXT_A$ and deletes the *random*-loop. The rule *clean* *deletes* an interval edge.

The control condition requires that the rule *start* is applied, initializing thereby the calculation process. The rule *calc* is applied as long as possible, calculating in this way an interval proportional to the pheromone value for each possible next edge, it has the length $f(\phi)$. Afterwards, the rule *rand* is applied, generating a random number between 0 and sum where sum is the sum of all generated distribution values. Based on the generated random value, the rule *choose* is applied to choose the interval where λ belongs. As a consequence of the right opening of the intervals, the intersection between each two different intervals is empty, therefore the choice is unambiguous. The rule *clean*

is applied with maximum parallelism, deleting all *interval*-edges in one step.

4.6 Summary

This chapter has introduced a graph-transformational approach to swarm computation, providing formal methods for the modeling of swarms and the analysis of their correctness and efficiency. In the first step, the main ideas of swarms and swarm computing were discussed. In this discussion, the major approaches to swarm computing were considered. The ideas were summarized with a graph transformational perspective, exploiting the concepts of graph transformation units and the massive parallelism of rule applications. Graph-transformational swarms and their computation were formally introduced.

The first examples, swarms in search for shortest paths and Hamiltonian cycles, indicate that graph problems can be solved efficiently and with high probability. Moreover they illustrate how several members that act locally can solve global problems.

In order to specify stochastic processes, the concept of control conditions was extended to specify matching based on distribution functions. The example of a simple pheromone-driven ant colony has shown how it is possible to use such control to model the stochastic behavior often required in swarm computing approaches.

The last section of this chapter discussed how graph-transformational swarms can be used in practice. It has proposed a way to regulate the applications of rules in swarm computations based on stepwise control, which is a concept based on finite-state automata. Furthermore, it has studied the implementation of the developed concept in the tool GrGen.Net. A concrete example is given and discussed. At the end, an example of a unit that implements a stochastic control was given.

Chapter 5

Unification capability

This chapter provides graph-transformational versions of three major swarm computing methods, namely ant colony optimization, particle swarm optimization and cellular automata. It demonstrates in this way the unifying capabilities of graph-transformational swarms. The three approaches were introduced in Chapter 2 in a general way. The current chapter recalls them more formally, starting by introducing some needed preliminaries in Section 5.1. Section 5.2 recalls the canonical version of particle swarm optimization and provides its graph-transformational counterpart. Section 5.3 discusses how to model a cellular automaton as graph transformational swarm. Section 5.4 surveys the colony optimization as a framework discussing its different components. It uses the traveling salesperson problem and the generalized assignment problem as illustrative examples and *ant system* and *MIN-MAX ant system* as illustrative versions. Finally, a graph transformational formulation is provided. The chapter is summarized in Section 5.6.

5.1 Preliminaries

Definition 25 (walk)

A *walk* in a graph $G = (V, E, s, t, l)$ is a sequence of nodes and edges $w = (v_1, e_1, v_2, \dots, v_k, e_k, v_{k+1})$, where $v_i \in V$, $e_i \in E$ and $s(e_i) = v_i$, $t(e_i) = v_{i+1}$ for $i = 1, \dots, k$ and $k \in \mathbb{N}_{>0}$. The nodes v_1 and v_{k+1} are called respectively the *start* and the *end* node of w . k is the *length* of the *walk*. The set of all walks in graph G is denoted W_G .

A walk can also be denoted by omitting the edges if it is obvious which edges are considered between two successive nodes in the walk. This is the case of the graphs considered in this chapter.

Definition 26 (cycle, Hamiltonian cycle)

A *cycle* in a graph G is a walk in G where the *start* node is equal to the *end* node. A *Hamiltonian cycle* in a graph $G = (V, E, s, t, l)$ is a cycle $(v_1, e_1, v_2, \dots, v_n, e_n, v_1)$ with $n = \#V$ containing every node in V and the nodes v_1, \dots, v_n are pairwise distinct (i.e., $v_i \neq v_j$ if $i \neq j$ for $i, j = 1, \dots, n$). The set of Hamiltonian cycles in a graph G is

denoted H_G

Given a finite set A , V^A denotes the set of nodes obtained by assigning every element $a \in A$ a unique node denoted v^a .

Definition 27 (complete bipartite graph)

Given two sets of nodes V_1 and V_2 , the *complete bipartite graph* $G_{V_1.V_2}$ is the graph where $V_{G_{V_1.V_2}} = V_1 \cup V_2$ and the set $E_{G_{V_1.V_2}}$ is the set of directed edges such that every pair in $V_1 \times V_2$ is connected by a unique pair of edges e_1, e_2 with opposite directions (i.e., $s(e_1) = t(e_2)$ and $t(e_1) = s(e_2)$).

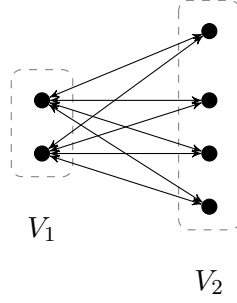


Figure 5.1 A complete bipartite graph connecting set V_1 (composed of the two nodes on the left side) with set V_2 (composed of the four nodes in the right side).

Definition 28 (complete graph)

A complete graph is a (*directed edge-labeled*) graph $G = (V, E, s, t, l)$ where every pair of distinct nodes is connected by a unique pair of edges e_1, e_2 with opposite directions (i.e., $s(e_1) = t(e_2)$ and $t(e_1) = s(e_2)$)

Definition 29 (discrete optimization problem)

A *discrete optimization problem* is a set of instances where every instance consists of a system (i, F, obj) where i is an entity that specifies a discrete search space, F is a set of feasible solutions and $obj : F \rightarrow \mathbb{R}$ is the objective function. It assigns a real value to every feasible solution. The goal is to minimize or to maximize the obj function.

5.2 Graph-transformational particle swarm

Particle swarm optimization is one of the major approaches to swarm intelligence one encounters in the literature in various variants (see, e.g., [48, 82, 83]). This section models a discrete version of particle swarm optimization in the framework of graph-transformational swarms.

A particle swarm acts in the Euclidean space \mathbb{R}^d for some dimension $d \in \mathbb{N}$. The space is provided with a fitness function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ and a neighborhood $N : \mathbb{R}^d \rightarrow \mathcal{P}(\mathbb{R}^d)$ (where $\mathcal{P}(X)$ denotes the power set of some set X). A swarm consists of n particles $i \in [n]$

each of which carries the following information at each time $t \in \mathbb{N}$: a *position* $p_{it} \in \mathbb{R}^d$, a *velocity* $v_{it} \in \mathbb{R}^d$, a *personal best (position)* $pb_{it} \in \mathbb{R}^d$, and a *best neighbor (position)* $bn_{it} \in \mathbb{R}^d$.

The initial positions p_{i0} and initial velocities v_{i0} are chosen randomly. The initial personal bests coincide with the initial positions, i.e. $pb_{i0} = p_{i0}$. In all steps, the best neighbor bn_{it} is the position of a particle j in the neighborhood of i , $p_{jt} \in N(p_{it})$, with maximum fitness, i.e. $f(p_{jt}) \geq f(p_{kt})$ for all $p_k \in N(p_{it})$. The positions, velocities and personal bests at time $t+1$ are given by the following formulas using the positions, velocities and personal bests at time t :

- $v_{i(t+1)} = v_{it} + U_t(0, \phi_1) \otimes (pb_{it} - p_{it}) + U_t(0, \phi_2) \otimes (bn_{it} - p_{it})$,
- $p_{i(t+1)} = p_{it} + v_{i(t+1)}$,
- $pb_{i(t+1)} = p_{i(t+1)}$ if $f(p_{i(t+1)}) > f(pb_{it})$ and $pb_{i(t+1)} = pb_{it}$ otherwise.

Here ϕ_1 and ϕ_2 are two pregiven bounds, $U_t(0, \phi_1)$ and $U_t(0, \phi_2)$ are vectors with randomly chosen components between 0 and ϕ_1 and ϕ_2 respectively, and \otimes is the componentwise product. A velocity represents a direction and a speed so that a particle moves in this direction with this speed from step to step where the velocity is adapted in such a way that the particle moves partly in the direction of the personal best and partly in the direction of the best neighbor. It is assumed that each particle is a neighbor of itself to guarantee that the best neighbor always exists. The goal is that one of the particles reaches a position, the fitness of which meets or exceeds a given bound. In the literature one can find a long list of examples of particle swarms which run successfully for a variety of optimization problems (see, e.g., [82]).

A simple way to discretize particle swarms is to assume that all position and velocity components and all randomly chosen scalars are integers. This discrete version of particle swarms can be transformed into the framework of graph-transformational swarms. Let PS be such a discrete particle swarm with the fitness function $f: \mathbb{Z}^d \rightarrow \mathbb{Z}$, the neighborhood $N: \mathbb{Z}^d \rightarrow \mathcal{P}(\mathbb{Z}^d)$, the bounds $\phi_1, \phi_2 \in \mathbb{N}$, the goal value $b \in \mathbb{Z}$, and n particles. Then the corresponding graph-transformational swarm is given in Fig. 5.2.

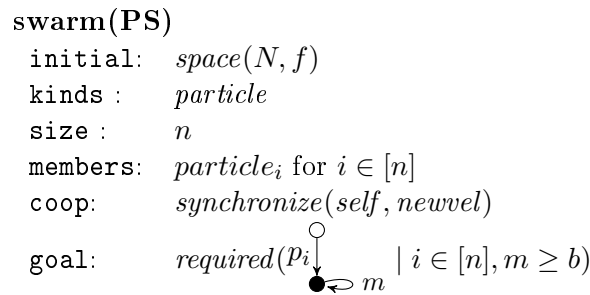
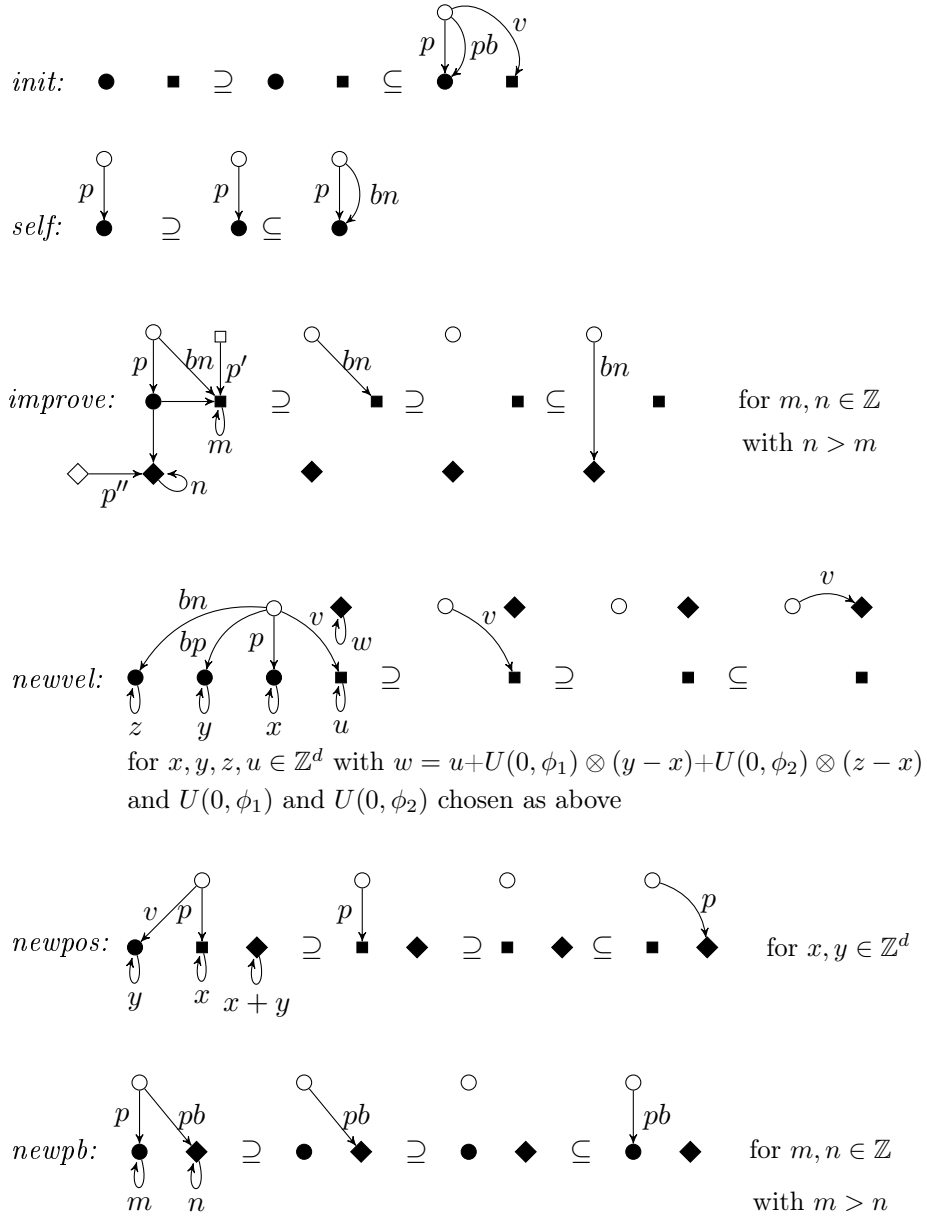


Figure 5.2 A graph transformational particle swarm

The initial environment graph is called $space(N, f)$ and has all points \mathbb{Z}^d in the d -dimensional Euclidean plane with integer coordinates as nodes. There is an unlabeled

edge (x, y) for $x, y \in \mathbb{Z}^d$ with the source x and the target y whenever $y \in N(x)$. Furthermore, each $x \in \mathbb{Z}^d$ has two loops $(x, 1)$ and $(x, 2)$ where x is source and target. The label of $(x, 1)$ is also x , the label of $(x, 2)$ is $f(x)$. All particles are of the same kind, specified by the unit *particle* in Figure 5.3. (For technical simplicity, it is assumed that $d > 1$.)

particle
rules:



control: *init* ; (*self* ; *improve!* ; *newvel* ; *newpos* ; *try(newpb)*)*

Figure 5.3 The unit *particle*

The member $particle_i$ for $i \in [n]$ is obtained by indexing p, v, pb and bn with i . All other labels are kept variable with $p', p'' \in \{p_1, \dots, p_n\}$ in particular. Due to the control

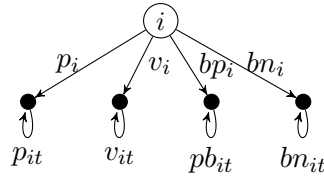
condition, the rule *init* is applied first and then never again. It chooses two points x and y , generates a new node (representing a particle) and two edges from this node to x , labeled with p and pb respectively, and an edge to y labeled with v choosing randomly an initial position, which is also the personal best, and an initial velocity. As nothing is removed, each two applications of *init* are parallel independent such that all particles can be initialized simultaneously. Afterwards, a sequence of rule applications is iterated starting with *self* followed by *improve* as long as possible. The application of *self* takes the current position as best neighbor by adding a *bn*-edge parallel to the *p*-edge. The rule *improve* can be applied if one can find a particle in the neighborhood with a better fitness. Applied as long as possible, the *bn*-edge points to the current best neighbor.

If now *newvel* and then *newpos* are applied, then the velocity and position of a particle are changed using the formulas above by redirecting the *v*-edge and *p*-edge accordingly. If the new position has a better fitness than the former personal best, then the rule *newpb* can be applied to update the personal best. The control condition *try(newpb)* requires that *newpb* is applied if possible.

The rules *self* can be applied to all particles in parallel as again nothing is removed. Two applications of *improve* for different particles are parallel independent as only the different *bn*-edges are redirected. Therefore, the improvements can be done in parallel provided that at most one *improve*-rule per particle is applied. The cooperation condition requires that the applications of *newvel* are synchronized, which means that they are done in parallel after all improvements have been performed. Each two applications of *newvel* for different particles are parallel independent as only different edges are redirected. Because of the same reason, all particles can get a new position by applying the *newpos*-rules in parallel. And analogously the *newpb*-rules can be applied in parallel afterwards as far as they are applicable at all. The cooperation condition requires that *self* is synchronized, which means that in the next round all applications of *self* start simultaneously. The goal requires that one of the particles reaches a position the fitness of which meets or exceeds the bound value b .

The rules *improve*, *newvel*, *newpos* and *newpb* describe how the attributes of a particle can be changed by redirecting the respective edges where the positive context (placed left-most) provides the parameters that must be considered in each case.

By definition, a run ρ of a particle swarm is determined by the choices of p_{i0} and v_{i0} for $i \in [n]$ and the vectors $U_t(0, \phi_1)$ and $U_t(0, \phi_2)$ for $t \in \mathbb{N}$. The family of quadruples $s_t = ((p_{it}, v_{it}, pb_{it}, bn_{it}))_{i \in [n]}$ may be seen as the swarm state at time $t \in \mathbb{N}$. Such a state can be transformed into a graph $gr(s_t)$ that has space (N, f) as subgraph and, for each $i \in [n]$, an additional node i as well as four new edges of the form



Consider, on the other hand, the computation of $swarm(PS)$ using the same choices as

the run ρ . Then the considerations of this section show that, for each $t \in \mathbb{N}$, the graph $gr(s_t)$ is computed after all *improve*-steps in round t through the iteration in the control condition. This proves the following correctness result.

Theorem 2

Let PS be a discrete particle swarm and $swarm(PS)$ the corresponding graph-transformational swarm. Then the results of runs in PS and the computations in $swarm(PS)$ are identical.

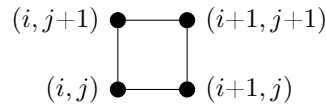
The proposed discretization can be applied to solve problems defined over continuous domains. In the literature there is also the trend to adapt *particle swarm optimization* to solve discrete optimization problems. The particles in this case correspond to problem solutions and the velocity and position updates, as introduced above, are redefined to be applicable to the discrete space (see i.e., [11, 70]). $swarm(PS)$ can also be adapted in the same way to solve discrete problems. In this case $space(N, f)$ and the operators in the rule *newvel* should be adapted to the corresponding domains. Despite those changes all other components can be used unchanged.

5.3 Graph-transformational cellular automata

Cellular automata are computational devices with massive parallelism that have been known for many decades (see, e.g., [12, 47, 101, 102, 104]). They are also considered as typical representatives of swarm computation (cf.[50]). In this section, we embed cellular automata into the framework of graph-transformational swarms.

A cellular automaton is a network of cells where each cell has got certain neighbor cells. A configuration is given by a mapping that associates a local state with each cell. A current configuration can change into a follow-up configuration by the simultaneous changes of all local states. The local transitions are specified by an underlying finite automaton where the local states of the neighbor cells are the inputs. If the network is infinite, one assumes a particular sleeping state that cannot change if all input states of neighbor cells are also sleeping. Consequently, all follow-up configurations have only a finite number of cells that are not sleeping if one starts with such a configuration.

To keep the technicalities simple, we consider 2-dimensional cellular automata, the cells of which are the unit squares in the Euclidean plane



for all $(i, j) \in \mathbb{Z} \times \mathbb{Z}$ and can be identified by their left lower corner. The neighborhood is defined by a vector $N = (N_1, \dots, N_k) \in (\mathbb{Z} \times \mathbb{Z})^k$ where the neighbor cells of (i, j) are given by the translations $(i, j) + N_1, \dots, (i, j) + N_k$. If one chooses the local states as colors, a cell with a local state can be represented by filling the area of the cell with the corresponding color. Accordingly, the underlying finite automaton is specified by a finite set of colors, say $COLOR$, and its transition $d: COLOR \times COLOR^k \rightarrow COLOR$.

Without loss of generality, we assume $white \in COLOR$ and use it as sleeping state, i.e. $d(white, white^k) = white$. Under these assumptions, a configuration is a mapping $S: \mathbb{Z} \times \mathbb{Z} \rightarrow COLOR$ and the follow-up configuration S' of S is defined by

$$S'((i, j)) = d(S((i, j)), (S((i, j) + N_1)), \dots, S((i, j) + N_k)).$$

If one starts with a configuration S_0 which has only a finite number of cells the colors of which are not $white$, then only these cells and those that have them as neighbors may change the colors. Therefore, the follow-up configuration has again only a finite number of cells with other colors than $white$. Consequently, the simultaneous change of colors of all cells can be computed. Moreover, there is always a finite area of the Euclidean plane that contains all changing cells. In other words, a sequence of successive follow-up configurations can be depicted as a sequence of pictures by filling the cells with their colors.

Example. The following instance of a cellular automaton may illustrate the concept. It is called *SIER*, has two colors, $COLOR = \{white, black\}$, and the neighborhood vector is $N = ((-1, 0), (0, 1))$ meaning that each cell has the cell to its left and the next upper cell as neighbors. The transition of *SIER* changes $white$ into $black$ if exactly one neighbor is $black$, i.e. $d: COLOR \times COLOR^2 \rightarrow COLOR$ with $d(white, (black, white)) = d(white, (white, black)) = black$ and $d(c, (c_1, c_2)) = c$ otherwise.

If one starts with the configuration S_0 with $S_0((10, 0)) = S_0((0, 10)) = S_0((30, 0)) = S_0((0, 40)) = black$ and $S_0((i, j)) = white$ otherwise, then one gets the configuration in Figure 5.4 after 50 steps.

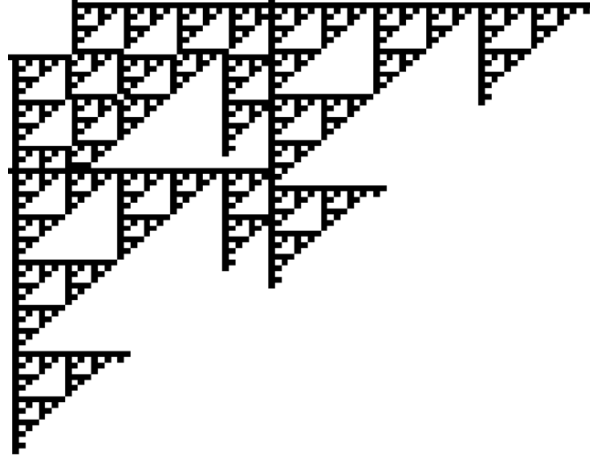
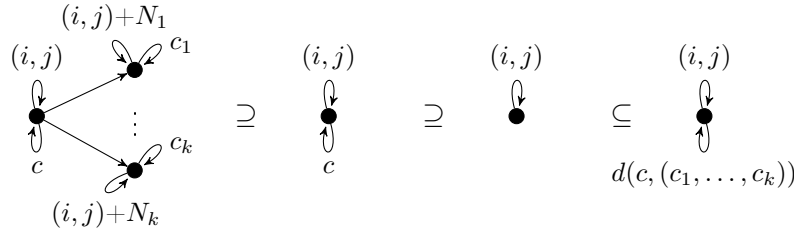


Figure 5.4 A pictorial representation of the configuration S_{50}

Starting with a single *black* cell, *SIER* iterates the Sierpinski gadget [cf., e.g., 76]. Cellular automata can be considered as graph-transformational swarms. Let CA be a cellular automaton with the neighborhood vector

$$N = (N_1, \dots, N_k) \in (\mathbb{Z} \times \mathbb{Z})^k,$$

the set of colors $COLOR$ and the transition function $d: COLOR \times COLOR^k \rightarrow COLOR$. Then a configuration $S: \mathbb{Z} \times \mathbb{Z} \rightarrow COLOR$ can be represented by a graph $gr(N, S)$ with the cells as nodes, with an unlabeled edge from each cell to each of its neighbors and two loops at each cell where one loop is labeled with the color of the cell and the other loop with the coordinates of the cell. The set of all these graphs is denoted by $\mathcal{G}(CA)$. If the color of a cell (i, j) changes, i.e. $d(S((i, j)), (S((i, j) + N_1), \dots, S((i, j) + N_k))) \neq S(i, j)$, then the following rule with positive context



can be applied to the node (i, j) in $gr(N, S)$ provided that $c = S(i, j)$ and $c_p = S((i, j) + N_p)$ for $p = 1, \dots, k$. Due to the loops that identify the nodes, the matching is unique and the matches of the left-hand sides of each two of such applicable rules do not overlap. Consequently, all those applicable rules can be applied in parallel yielding $gr(N, S')$ where S' is the follow-up configuration of S . This remains true if the (empty) sleeping rule is applied to each other node because it is always applicable, is always independent of each other rule application and does not change the result. In other words, the derivation step $gr(N, S) \Longrightarrow gr(N, S')$ is a swarm computation step if the rules above belong to members of a swarm which can be defined as follows:

swarm(CA)
 initial: $\mathcal{G}(CA)$
 kinds : $gtu(P((0, 0)))$
 size : $\mathbb{Z} \times \mathbb{Z}$
 members: $gtu(P((i, j)))$ for $(i, j) \in \mathbb{Z} \times \mathbb{Z}$
 coop: *free*
 goal: *all*

where the kind and the members are units induced by the sets of rules $P((i, j))$ containing all rules above for $(i, j) \in \mathbb{Z} \times \mathbb{Z}$ and the transition d . Every member $gtu(P((i, j)))$ is obtained from the kind $gtu(P((0, 0)))$ by translating all points in the plane by (i, j) which is a special relabeling. Conversely, a computation step $gr(N, S) \Longrightarrow H$ in $swarm(CA)$ changes a c -loop into a $d(c, (c_1, \dots, c_k))$ -loop at the node with the (i, j) -loop if and only if, for $l = 1, \dots, k$, the neighbor with the $(i, j) + N_l$ -loop also has a c_l -loop. All other c -loops are kept. This means that $H = gr(N, S')$. In summary, each cellular automaton can be transformed into a graph-transformational swarm such that the following correctness result holds.

Theorem 3

Let CA be a cellular automaton with neighborhood vector N and let $swarm(CA)$ be

the corresponding graph-transformational swarm. Then there is a transition from S to S' in CA if and only if $gr(N, S) \Longrightarrow gr(N, S')$ is a computation step in $swarm(CA)$.

Therefore, cellular automata behave exactly as their swarm versions up to the representation of configurations as graphs. We have considered cellular automata over the 2-dimensional space $\mathbb{Z} \times \mathbb{Z}$. It is not difficult to see that all our constructions also work for the d -dimensional space \mathbb{Z}^d in a similar way. One may even replace the quadratic cells by triangular or hexagonal cells.

5.4 Ant colony optimization as a framework

The current section shows how it is possible to integrate the metaheuristic ant colony optimization (ACO) into the graph-transformational swarms framework. While Section 2.2 introduces ACO in a general way, considering its inspiration and its general functioning, this section focuses on its formal description. The metaheuristic was proposed in 1999 [20], however a formal formulation has only been given in 2004 [22]. This was a reaction to the wide range of developed versions each of which was adapted to solve a given optimization problem. The formulation in 2004 is proposed to unify the different existing ACO approaches as a framework. A detailed description of the ACO-framework can be found in [22]. This section summarizes¹ that description.

The framework of ACO is considered to be a composition of three components: (1) *ACO-input* is a system that represents an instance of a given problem in a way that it can be solved by the metaheuristic, (2) *ACO-procedures* are the procedures that the metaheuristic iterates until a goal is met and (3) *ACO-parameters* are parameters and functions that specify a complete algorithm.

The mapping from a minimization problem to an ACO-input is illustrated via the traveling sales person (TSP) and the generalized assignment problem (GAP). The ACO-parameters are specified for two well established versions, the *Ant System* (AS) and *MAX-MIN Ant System* (MMAS).

In the second part, the ACO-methaheuristics is formulated as a graph transformational swarm. Furthermore, the components of the swarm for the two problems GAP and TSP as well as the algorithm MMAS are specified.

The proposed graph transformational swarm is composed of three kinds: *ant*, *updater* and *best*. The ant members walk on the graph, building their solutions in parallel. After that, the members of kind *updater* update the values of the pheromone. An iteration is achieved by choosing the best solutions through the members of kind *best*. The swarm repeats this iterations until the goal, which consists of finding a solution with a cost less or equal a value b , is met.

Figure 5.5 displays an overview of the ACO-framework and the relationship between its components. These are described as follows:

¹The summary is performed with some adaptation in the formulations used in the original formulation. The main difference consists of the definition of the minimization problem. In the original version, the authors use a constraints-based definition. Here, a classical one based on the specification of the set feasible solution is used. Another difference is the decomposition into three components. This allows a better description of the attempted unification using graph class expressions.

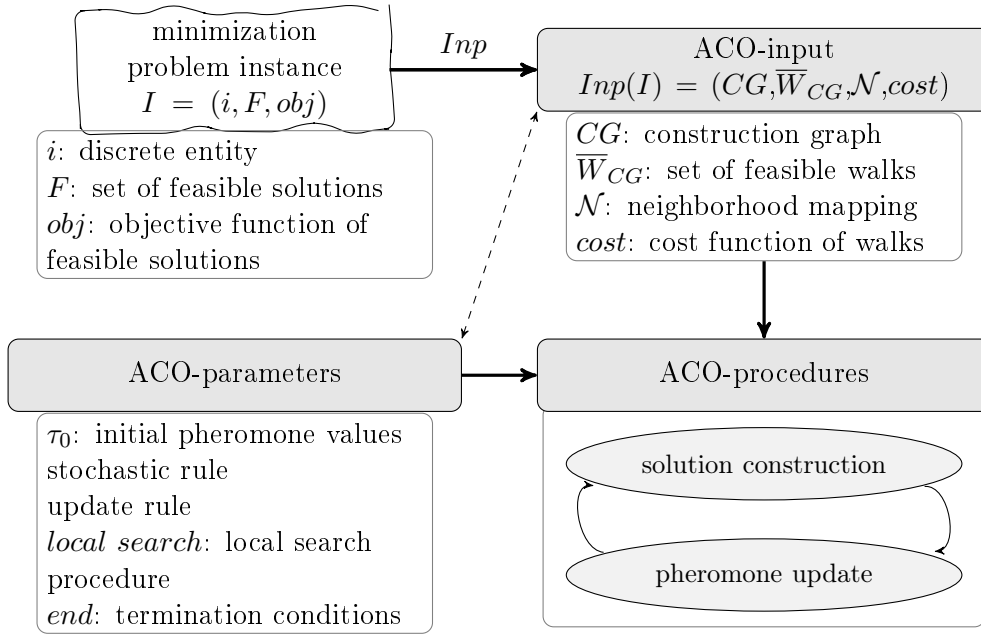


Figure 5.5 Overview of the ant colony optimization framework

5.4.1 ACO-input

In order to apply ACO to a given minimization problem, one has to define a mapping Inp that assigns to every instance $I = (i, F, obj)$ a model $Inp(I)$ called *ACO-input*. The main component to be specified in the *ACO-input* is the *construction graph* CG that encodes the corresponding entity i . The edges of the construction graph are labeled by a set of attributes. For a given subset of edges one of these attributes should be the pheromone value τ . τ is initialized by the algorithm and is updated during the computations. The other attributes are problem specific and are invariant during the computations. An equivalent to feasible solutions should be specified in the graphical setup in term of *feasible walks*. The set of all feasible walks in a construction graph CG is denoted \overline{W}_{CG} . Another important characteristic of ACO is the incremental construction of solutions. For this purpose a mechanism is needed to guide the ants in their decision in such a way that only feasible walks are constructed at the end of an iteration². That is, a kind of mechanism that transforms the specification from the global level (the feasible solution) to the local level (the decision of an ant in every step of its movement in the underlying graph). ACO proposes the notion of *feasible neighborhood*. Given a walk w the feasible neighborhood $\mathcal{N}(w)$ defines the set of possible nodes that can be added to w . In other terms, if an ant has constructed a walk $w = (v_1, \dots, v_{k+1})$ and is now situated at the (end) node v_{k+1} , then $\mathcal{N}(w)$ is the set of adjacent nodes to v_{k+1} that can be visited and added to the walk w . In this way, the feasible neighborhood mapping restricts the ants to building only walks that belong to a set $\overline{W}_{CG}^{partial} \subset \overline{W}_{CG}$. Consequently, $\overline{W}_{CG}^{partial}$ is

²In some cases this constraint can be relaxed allowing ants to also construct non-feasible solutions. Such solutions are then adapted in a suitable way to become feasible solutions

called the set of *feasible partial walks*. Furthermore, the objective function should also be adapted to apply to walks. The adapted objective function is denoted *cost*.

More formally, given an instance $I = (i, F, obj)$ of a discrete minimization problem, the corresponding ACO-input is a system $Inp(I) = (CG, \overline{W}_{CG}, \mathcal{N}, cost)$ where

- CG is a graph called *construction graph*. It is a presentation of the entity i as a graph. The nodes are called *solution components*.³ The label function $l_{CG} : E_{CG} \rightarrow \Sigma$ assigns a vector of attributes (a_1, \dots, a_k) with $k \in \mathbb{N}$ to every edge. For a subset of edges $E_\tau \subset E_{CG}$, the first attribute is reserved to the pheromone value,
- $\overline{W} \subseteq W_{CG}$ is the set of *feasible walks* in CG . Usually, every *feasible walk* corresponds to a feasible solution in F ,
- $\mathcal{N} : W_{CG} \rightarrow \mathcal{P}(V_{CG})$ is the *neighborhood* mapping, where $\mathcal{P}(V_{CG})$ is the power set of V_{CG} containing all its subsets. Given a walk $w \in W_{CG}$ the mapping \mathcal{N} returns a subset of nodes in V_{CG} and
- $cost : \overline{W} \rightarrow \mathbb{R}$ is the *cost* function of a given feasible walk.⁴

ACO-input for TSP

In TSP, the search space of an instance is specified by a pair $(G, dist)$ where G is a complete⁵ graph with n nodes and $dist : E_G \rightarrow \mathbb{R}^+$ assigns to every edge a weight called *distance*. The set of feasible solutions F corresponds to the set of *Hamiltonian cycles* H_G . The objective function of a hamiltonian cycle is the sum of distances in its edges. The goal is to find a hamiltonian cycle with minimal sum. An instance corresponds to a system $I_{TSP} = (i, F, obj)$ where,

- $i = (G, dist)$
- $F = H_G$
- $obj(v_1, \dots, v_n, v_1) = \sum_{i=1}^{n-1} dist(v_i, v_{i+1}) + dist(v_n, v_1)$
with $(v_1, \dots, v_n) \in H_G$.

Given an instance $(G, dist)$ of TSP, the corresponding ACO-input is the system $(CG, \overline{W}, \mathcal{N}, cost)$ where:

- $CG = (V_G, E_G, s_G, t_G, l)$ with $l = (\tau, dist)$
- $\overline{W} = H_G$

³In the literature the construction graph CG is usually defined to be fully connected. For more flexibility, this restriction is suspended in this formulation.

⁴In some cases *cost* can be defined over the set W offering an incremental computation.

⁵Not complete graphs can be transformed to fully connected graphs by connecting the unconnected nodes with edges having a very large distance. During an optimization process such large distances are unattractive and the corresponding edges are disqualified from being chosen.

- $\mathcal{N}((v_1, \dots, v_k)) = \begin{cases} v_1 & \text{if } k = n \\ \{V_G \setminus \{v_1, \dots, v_k\}\} & \text{if } k = 1, \dots, n-1 \end{cases}$
- $cost(w) = obj(w)$ for $w \in \overline{W}$

The construction graph is equal to the graph G up to its labeling. The only modification is the definition of the pheromone value as the first component of the attributes vector of edges, the second attribute is the value of the *dist* function. The set of feasible walks is the set of Hamiltonian cycles. The *feasible neighbor* of walk w comprising all nodes of the graph is the start node of the walk. For other cases, the feasible neighbors are all nodes which have not yet been visited (do not belong to w). The cost of a walk corresponds to the value of its objective function.

ACO-input for GAP

In *GAP* the search space of an instance is specified by a system (A, T, cp, r, c) where A is a finite set of agents, T is a finite set of tasks, $cp : A \rightarrow \mathbb{N}$ assigns *capacity* of resources to every agent a , $r : T \times A \rightarrow \mathbb{N}$ assigns to every pair $(t, a) \in T \times A$ the amount of resources consumed by the task t if it is assigned to the agent a , $c : T \times A \rightarrow \mathbb{N}$ assigns the cost of assigning a task $t \in T$ to an agent $a \in A$. Feasible solutions are assignments where every task is assigned to a unique agent (Equation 5.2) without exceeding the capacities of the agents (Equation 5.1). An assignment can be specified by a mapping $y : T \times A \rightarrow \{0, 1\}$ where $y(t, a) = 1$ if the task t is assigned to the agent a and $y(t, a) = 0$ otherwise. The objective function value of a feasible solution y is the sum of the assignment costs. The goal is to minimize the objective function.

An instance of *GAP* corresponds to a system $I_{GAP} = (i, F, obj)$ where:

- $i = (A, T, cp, r, c)$;
- $F = \{y : T \times A \rightarrow \{0, 1\}\}$ with

$$\sum_{t \in T} r(t, a) \cdot y(t, a) \leq cp(a), \quad a \in A \quad (5.1)$$

$$\sum_{a \in A} y(t, a) = 1, \quad t \in T \quad (5.2)$$

- $obj(y) = \sum_{t \in T} \sum_{a \in A} c(t, a) \cdot y(t, a)$.

The following translation uses a model based on the work in [66]. Given an instance $I_{GAP} = ((A, T, cp, r, c), F, obj)$, the corresponding ACO- *input* is given by the system $(CG, \overline{W}, \mathcal{N}, cost)$ where:

- CG is the construction graph which is built in two steps. First, the complete bipartite graph over the sets V^T and V^A is generated. Afterwards and in order to encode the capacities of agents, every agent node is decorated with a loop

labeled with the corresponding capacity. The set of all such loops is denoted E_{VA} . The edges from V^T to V^A are labeled with the values of the vector function (τ, d) where τ is the pheromone function initialized by a given value and d is the distance function between a task- and an agent-node, it is equal to the c function between the corresponding elements in T and A . That is,

$$E_{CG} = \{E_{VA} \cup E_{G_{V^T.V^A}}\}$$

and

$$l(e) = \begin{cases} cp(s(e)) & \text{for } e \in E_A \\ (\tau(e), d(e) = c(s(e), t(e)), r(e) = r(s(e), t(e))) & \text{for } e \in E_{G_{V^T.V^A}} \end{cases}$$

- \overline{W} is the set of feasible walks corresponding to all walks starting with a task node and alternating the visit between agent nodes and task nodes. All task nodes should be visited once at the end. i.e.,

$$\overline{W} = \{(v^{t_1}, v^{a_1}, \dots, v^{t_{\#T}}, v^{a_{\#T}})\}$$

with $v^{t_i} \in V^T$, $v^{a_i} \in V^A$, $i \in [\#T]$ and if $i \neq j$ then $v^{t_i} \neq v^{t_j}$ for $i, j \in [\#T]$

- \mathcal{N} is the neighborhood mapping. It differentiates between two cases. If the last node in the walk is a task node, the next node has to be an agent node in such a way that the capacity of the corresponding agent is not exceeded. If the last node in the underlying walk is an agent node, the next node should be a task node that is not yet visited. That is,

$$\mathcal{N}((v^{t_1}, v^{a_1}, \dots, v^{t_k}, v^{a_k}, v^{t_{k+1}})) = \{v^a \in V^A \mid \sum_{j=1, a_j=a}^k r(t_j, a) + r(t, a) \leq cp(a)\} \quad (5.3)$$

and

$$\mathcal{N}((v^{t_1}, v^{a_1}, \dots, v^{t_{k-1}}, v^{a_{k-1}}, v^{t_k}, v^{a_k})) = \{V^T \setminus \{v^{t_1}, \dots, v^{t_k}\}\} \quad (5.4)$$

for $t_i \in T$, $a_i \in A$, $i = 1, \dots, k$ and $k = 1, \dots, \#T-1$

- $cost$ is the cost function of feasible walks. It assigns to every feasible walk w the objective value of the corresponding assignment y_w .

$$cost(w) = cost(y_w)$$

$$\text{with } w \in \overline{W} \text{ and } y_w(e) = \begin{cases} 1 & \text{if } e \in w \\ 0 & \text{otherwise} \end{cases}$$

5.4.2 ACO-procedures

Given an ACO-input, the metaheuristic ACO consists of the iteration of two procedures:⁶

1. **Solutions construction:** In this procedure each ant in the swarm attempts to incrementally construct a walk that corresponds to a solution of the underlying problem. Starting in a randomly chosen node with an empty *walk* an ant repeats a movement step until it constructs a feasible *walk* or meets another end condition. The state of an ant is given by its memorized nodes visited so far as a *walk* (v_1, \dots, v_k) which includes the current position v_k . The movement step consists of choosing stochastically a *feasible neighbor* v_{next} in $\mathcal{N}((v_1, \dots, v_k))$ which becomes the new position. v_k is added according to its current *walk* yielding to the new *walk* $(v_1, \dots, v_k, v_{next})$. The stochastic choice of solution components from feasible neighbors is based on pheromone values and other attributes of the outgoing edges from v_k to all feasible neighbors.
2. **Pheromone update:** The values of pheromone of the edges are changed, taking into consideration the quality of the solutions produced so far. Usually the pheromone values of the edges that belong to the best walks are increased. In addition, the evaporation behavior of the real chemical substance is imitated by decreasing the values of all pheromone in the construction graph.

The two procedures are iterated until a termination condition is met. In the following, it is considered that a bound b is given. That is the computation ends if an ant constructs a solution with a *cost* lower or equal b .

Remark 5

The ants use the neighborhood mapping \mathcal{N} to choose the next node in every step. In this way \mathcal{N} forces ants to construct walks belonging to $\overline{W}^{partial}$. Therefore the domain of \mathcal{N} can be restricted to $\overline{W}^{partial}$.

The next subsection illustrates the ACO-parameters for the two algorithms *ant system* and *MAX-MIN ant system*.

5.4.3 ACO-parameters

As mentioned above, there are several methods that belong to the ACO family. They all follow the procedures introduced in the previous subsection but they differ in some specifications such as the way in which the pheromone values are updated, or what the stochastic rule followed by the ants in their movement steps looks like. In order to specify a complete implementation, additional parameters such as the number of ants or the initial pheromone values are needed. All these parameters are summarized into the components called *ACO-parameters* which comprises the following list of items amongst others:

⁶In the literature there is an additional procedure called local search. This procedure is optional and consists of improvements of the constructed solutions using classical local search methods.

- m : the number of ants. It is generally proportional to the size of the underlying construction graph,
- τ_0 the initial pheromone function that assigns an initial pheromone value to every edge,
- the *stochastic rule* is a rule that the ants follow in every step to choose the next node with a probability proportional to the pheromone values present in the corresponding edge,
- the *update rule* is responsible for the update of the pheromone values in every edge after all ants have constructed their walks. Usually the update takes into consideration the best solutions found so far and increases the pheromone values in the corresponding walks.

ACO-parameters in ant system

ant system (AS) is the first version of the ACO algorithms. It has been proposed by Dorigo et al. [21] to solve the *traveling salesperson problem* (TSP).

At the beginning, every edge of the *construction graph* is initiated with the same (virtual) pheromone value τ_0 . A number of ants m are randomly assigned to starting nodes.

- τ_0 is a constant function that assigns the same (virtual) pheromone value to every edge of the construction graph.
- *stochastic rule*: Given an ant with a walk w the probability to choose an edge from the feasible neighbors is given by the following equation.

$$p_w(e) = \frac{[\tau(e)]^\alpha \cdot [\eta(e)]^\beta}{\sum_{e \in N(w)} [\tau(e)]^\alpha \cdot [\eta(e)]^\beta} \quad \forall e \in N(w) \quad (5.5)$$

where α and β are two parameters that control the relative influence of the pheromone or the heuristic value η respectively. The heuristic value of an edge is problem specific. In the case of TSP , it is equal to the reciprocal value of its distance (i.e., $\eta(e) = \frac{1}{dist(e)}$).

- *update rule*: the pheromone value of an edge e is updated as follows:

$$\tau(e) \leftarrow (1 - \rho) \cdot \tau(e) + \sum_{a=1}^n \Delta\rho_e^a \quad (5.6)$$

where $\Delta\rho_e^a$ is the value of pheromone that ant a deposits on the edge e . It is given by:

$$\Delta\rho_e^a = \begin{cases} 1/cost(w_a) & \text{if edge } e \in w_a \\ 0 & \text{otherwise} \end{cases} \quad (5.7)$$

where w_a is the walk constructed by the ant a . In other words, only the pheromone values of the edges visited by the ants $a \in [n]$ are increased with an amount equal

to the inverse of the cost $cost(w_a)$ of the constructed walk w_a . The idea behind the equation 5.7 is that the edges belonging to walks with less cost receive more pheromone. By means of the probability equation in 5.5, these edges are more likely to be chosen by ants in the next iterations.

AS has been tested on small TSP instances (up to 70 nodes). The results have revealed that the algorithm can solve instances with up to 30 cities with acceptable performance, compared to other heuristics. However, it has shown its limit for larger instances [21]. This was a main factor for developing other *ACO* algorithms with better performances.

ACO-parameters in $\mathcal{MAX} - \mathcal{MIN}$ ant system

$\mathcal{MAX} - \mathcal{MIN}$ ant system (\mathcal{MMAS}) introduces modifications of the *Ant System* in the way in which the update of pheromone values is performed. The most important contribution of the method is the definition of a maximal and a minimal bound τ_{max} and τ_{min} for all pheromone values. These are calculated based on some heuristic information about the problem. This simple modification brings forth significant improvements of the convergence characteristics of *ACO* methods preventing rapid stagnation on sub-optimal solutions. The solution construction using the equation 5.5 as described above is kept the same. The elements of *ACO*-parameters that correspond to $\mathcal{MAX} - \mathcal{MIN}$ ant system are:

- $\tau_0 = \tau_{max}$,
- *stochastic rule*: the same as *AS* (see Equation 5.5),
- *update rule*: two main modification regarding the *AS* update are performed. First, only the edges belonging to the best (or to the best so far) cycle are enhanced. The second modification introduced by \mathcal{MMAS} consists of the definition of an interval $[\tau_{min}, \tau_{max}]$ to limit the range of pheromone values.

Accordingly, the update equation in \mathcal{MMAS} for a given edge e looks as follows:

$$\tau(e) \leftarrow [(1 - \rho) \cdot \tau(e) + \Delta\rho(e)]_{\tau_{min}}^{\tau_{max}} \quad (5.8)$$

where $\Delta\rho(e) = 1/cost_b$ if e belongs to the best *walk* or 0 else. ρ is a parameter called the evaporation rate. The notation $[\]_{\tau_{min}}^{\tau_{max}}$ means that the value between the brackets is replaced by τ_{max} if it exceeds τ_{max} or by τ_{min} if it falls below τ_{min} .

In addition, \mathcal{MMAS} has proposed many other technical improvement like the reinitialization of the bounds τ_{max} and τ_{min} if the algorithm stagnates.

In the following section a graph transformational swarm version of the *ACO* metaheuristic is given.

5.5 Graph-transformational ant colony

This section gives a graph transformational version of the *ACO* metaheuristic.

```

swarm(ACO)
  initial:  edge-id & best(CG)
  kinds :   ant, updater, best
  size :    m, p, 1
  members:  anti, updaterj, best for  $i \in [m], j \in [p]$ 
  coop:     (ant; best; updater)*
  goal:     (  $\begin{array}{c} \diamond \\ \uparrow \\ \text{best}, \text{cost}_b \end{array} \mid \text{cost}_b \leq b$  )

```

Figure 5.6 The schematic representation of the graph transformational swarm ACO

Given an ACO-input $(CG, \overline{W}, \mathcal{N}, cost)$ a graph transformational swarm version of the ACO metaheuristic is specified in Figure 5.6.

The initial environment corresponds to *edge-id* & *best(CG)* graph. It is obtained by performing two modifications on the construction graph CG . In order to encode the best solution, an isolated node called *best*-node is added. The *best*-node has a loop labeled with *best*, σ where σ is a very large positive real number that initializes the cost value of the best feasible walk found so far. To allow parallel computations based on the notion of stationary members, *identifier* edges are created. Every pheromone edge in CG becomes a parallel edge having the same direction labeled by an identifier. Let us assume that the number of pheromone edges in the underlying graph is equal to p . Thus, the swarm has p members of kind *updater* such that each one is assigned to a unique identifier edge. The remaining kinds are *ant* and *best*. While there are m members of kind *ant* in the swarm, only a unique member of kind *best* exists in the swarm.

The cooperation condition requires that the ants members perform their rules until all ants fall asleep (beside the sleep rule, no other rule is applicable), followed by the best member and at the end the updater members. This iteration is arbitrary often repeated. The goal consists of finding a solution with a cost denoted $cost_b$ that is lesser or equal to the given bound b .

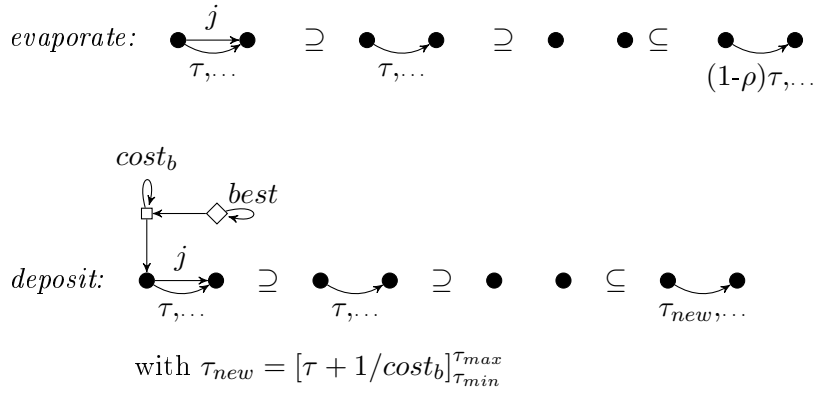
swarm(ACO) models the *ACO – procedures* in the following way: The solution construction procedure is realized through the members of kind *ant*, and the update procedure is effectuated via m members of kind *updater*. However, in order to specify the kinds a set defined through *ACO – parameters* is needed. In the following, *MMAS* as one of the most widespread variants of ACO is considered as basis for the specification of the kinds. The proposed graph transformational version can be considered as a parallel version of the original one. More details about parallelism follow below. Furthermore the exact behavior of ants also depends on the optimization problem into consideration. The kinds *updater* and *best* are rather problem independent and are therefore specified first.

The kinds updater and best In the considered swarm the stationary⁷ members of kind *updater* specified in Figure 5.7 perform the update in Equation 5.8 for all pheromone edges in parallel. The rules perform changes on the pheromone value τ of the pheromone edge, let's say e_j , identified by the parallel identifier j -edge. τ is the first component in the label vector of e denoted $\langle \tau, \dots \rangle$ meaning that the other components are problem specific and are not changed by the updater rules.

The rule *evaporate* decreases the pheromone τ of the edge e_j by an amount $\rho \cdot \tau$ where ρ is the evaporation rate as introduced in Equation 5.8. The rule *deposit* augments the pheromone value τ of the edge e_j by $\frac{1}{cost_b}$ provided that the edge e_j belongs to the best walk found so far with the cost $cost_b$. The resulting value is replaced by τ_{max} if it is greater than τ_{max} or by τ_{min} if it is lower than τ_{min} . The control condition requires that the evaporation is applied first followed by *deposit* if it is applicable.

updater

rules:



control: *evaporate*; *try(deposit)*

Figure 5.7 The kind *updater*

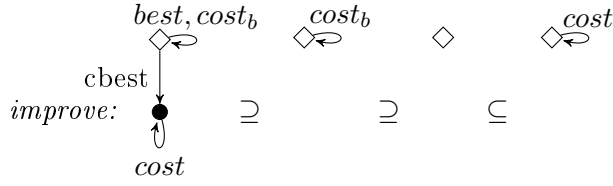
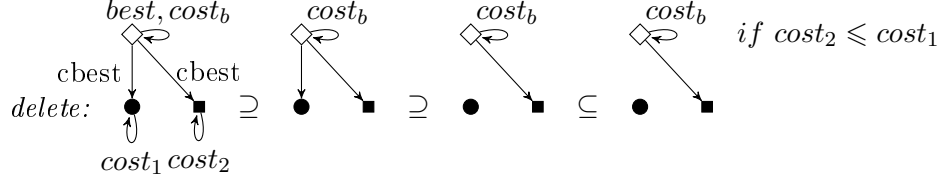
The kind *best* is specified in Figure 5.8. In the chosen approach, it is considered that the ants "work" in parallel. It is therefore possible that more than one ant consider its constructed walk as the best one if its cost is strictly lesser then the cost of the best walk found so far. Such a walk is considered then to be solely as a candidate to be the best walk and the ants create an edge from the best node to the corresponding *walk*-node by a *cbest*-edge (for more details see the description of the kinds ant in following section). The role of the *best* member which is equal to its kind is to choose the best walk found in the current iteration. This is performed by two rules. Given two candidates for the best solution, the rule *delete* deletes the one with the greater cost. The rule *delete* is applied until only one candidate best walk is leftover. The rule *improve* updates the cost value

⁷stationary members are introduced in Section 6.2.

of the best walk found so far making it equal to the cost of remaining candidate solution after the application of the *delete* rules.

best

rules:

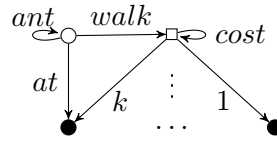


control: *delete!* ; *improve*

Figure 5.8 The kind *best*

The kind ant The specification of the kind ant depends on the optimization problem taken into consideration. In the current paragraph, two examples of the kind ant are given. The kinds *ant(TSP)* and *ant(GAP)* which are specialized to solve respectively the traveling salesperson problem and the generalized assignment problem.

In contrast to the previous chapter where an ant is modeled by an edge, here an ant is modeled by a node called *ant*-node with a loop having its name as label and additional information that represent its current state. It can be depicted as follows:

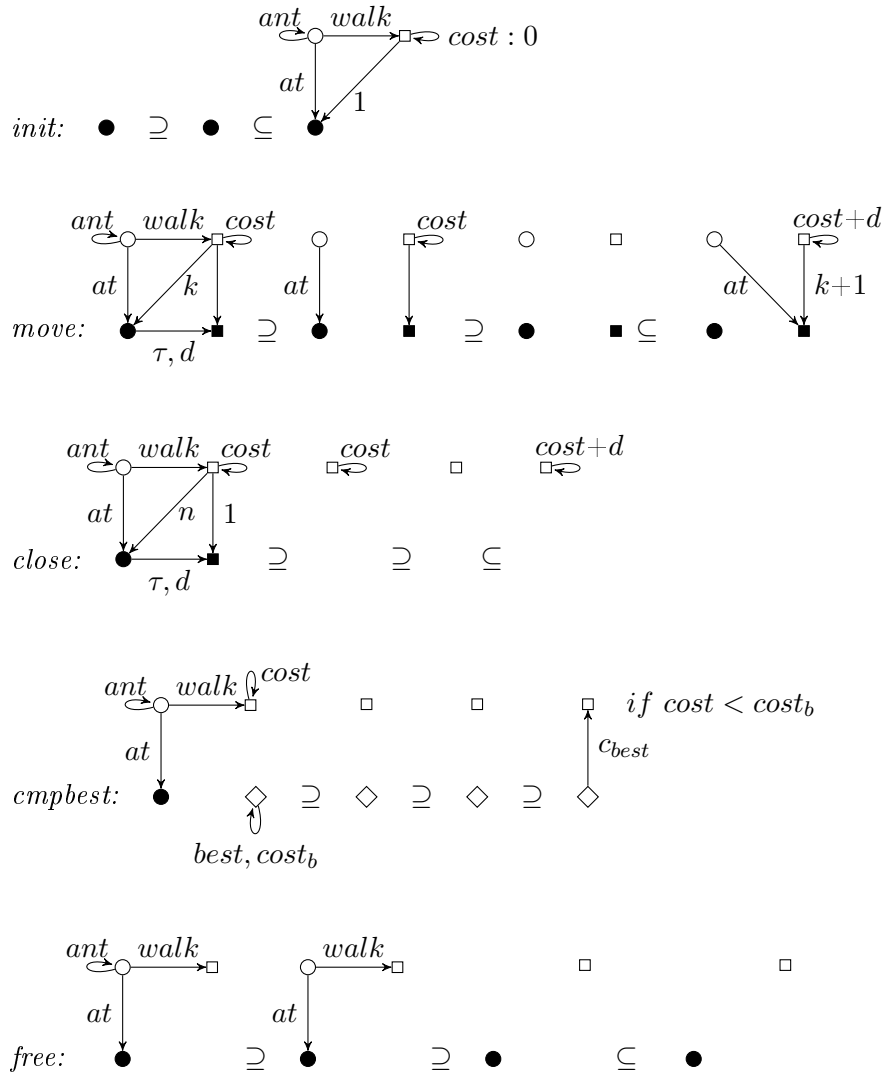


The current state of an ant corresponds to its current location (modeled by an outgoing *at*-edge having the current node as target) and the walk constructed so far represented by an outgoing *walk*-edge having as target a *walk*-node. A *walk*-node encodes a walk (v_1, \dots, v_k) by having an outgoing edge e_i to every node v_i labeled by i indicating its position in the walk for $i = 1, \dots, n$. The node that represents the current position coincides with the last node in the walk constructed so far. The loop in the *walk*-node represents the *cost* function. Its label is the value of the *cost* of the walk constructed so far.

Given an *ACO-input* of the traveling salesperson problem as introduced above, the kind that models the ant specialized to construct walks for this problem is called *ant(TSP)* and given in Figure 5.9.

ant(TSP)

rules:



control: *init*; $\text{move}_{\frac{\tau^\alpha}{d^\beta}}!$; *close*; *try(cmpbest)*; *free*

Figure 5.9 The kind *ant(TSP)* which is specialized to construct solutions for the traveling salesperson problem

At the beginning, the rule *init* initializes the iteration by creating an *ant*-node with

initial location (*at*) by choosing an arbitrary node in the graph. The corresponding walk has an initial cost equal to 0 and a start node that coincides with the current location. The rule *move* is applied as long as possible. It chooses a feasible neighbor that has not yet been visited from the underlying ant stochastically. The rule *move* makes the ant move one step forward to a feasible neighbor which it has not yet visited. The control condition requires that the choice is made stochastically using the function $\frac{\tau^\alpha}{d^\beta}$. The ant therefore changes its location (*at*) and updates its *walk* by linking the new location to the *walk* through an edge labeled $k + 1$ where k is the label of the location before applying the rule *move*. The value of the cost function *cost* is updated to $cost + d$ where d is the distance between the old and new location. The rule *close* closes the Hamiltonian cycle by adding the distance between the last and the first node in the *walk* to the cost value. After the construction of a *feasible walk* it is compared with the best solution found so far through the rule *cmpbest*. The rule is applied if the underlying cost value is smaller than the best walk so far. In this case it is considered as a candidate to be a best walk (*cbest*) and an edge labeled with *cbest* is added between the *best*-node and the constructed solution. It is solely a candidate because there may be other ants with better walks in the same iteration. At the end of the construction iteration the *ant*-node is deleted along with its outgoing edges.⁸

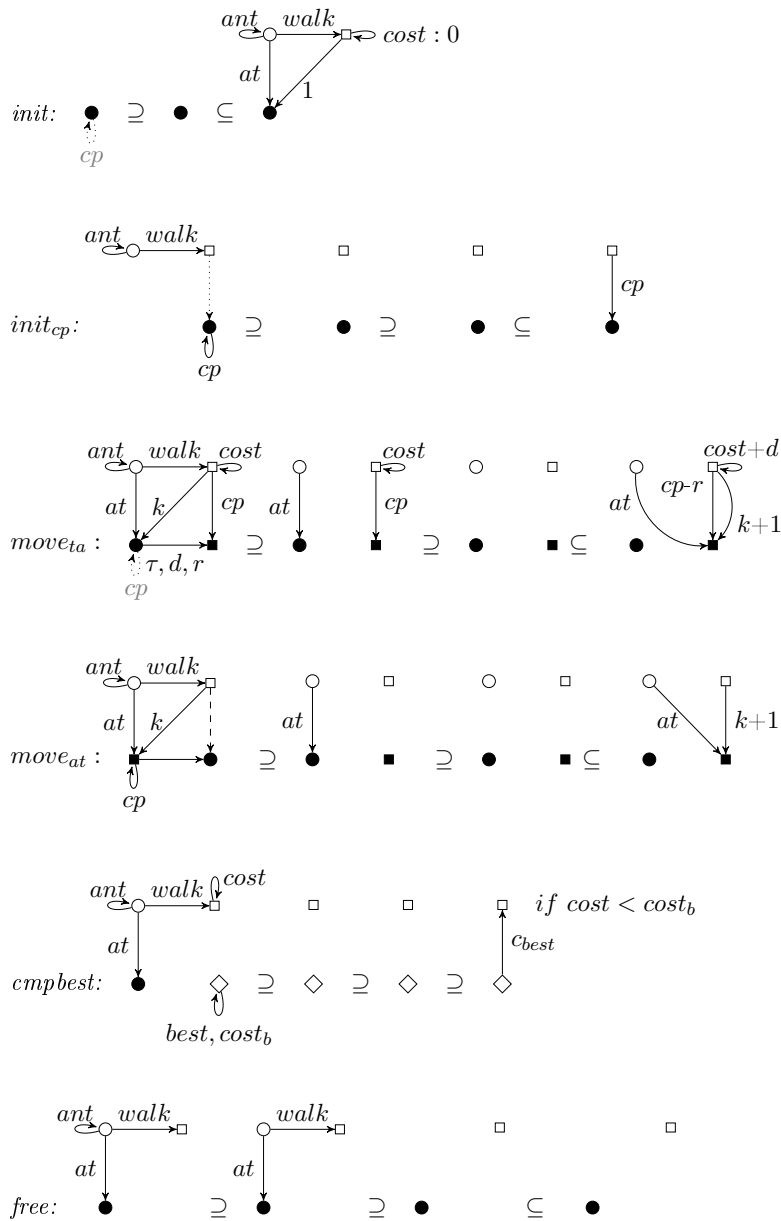
Given an *ACO – input* of the generalized assignment problem, Figure 5.10 specifies the kind *ant(GAP)* that is specialized to construct the corresponding feasible walks.

The *init* rule initializes the *ant* node with a walk of $cost=0$, starting in an arbitrary *task*-node. For this reason, the negative application condition requires that the initial node should have no capacity loop which is a characteristics only of *agent*-nodes in the underlying graph. In order to encode the change of capacity consumed during the assignment process, the maximal parallel application of *init_{cp}* create an edge from the *walk*-node to every agent node labeled by the capacity of the the underlying agent. After the initialization process, the control condition requires that one of the movement rules: *move_{ta}* or *move_{at}* should be stochastically applied. The both rules make *ant* forward moving, changing its current location and creating an *order*-edge to the new location. The movement step should be applied as long as possible. *move_{ta}* models the movement from a task node to an agent node. It is applied when the ant is located (*at*) a *task*-node (i.e a node that has no a capacity loop). The rule chooses stochastically an agent with a capacity cp_1 such that the amount of resources consumed c of such assignment is lower than its current capacity, regarding the constructed walk $c \leq cp$. That is, the rule *move_{ta}* incorporates the constraint formulated in the Equation 5.1. In addition to the forward movement of *ant*, the application of the rule *move_{ta}* updates the remaining capacity of the chosen agent. The rule *move_{at}* is applied when *ant* is situated at an *agent*-node (having a capacity-loop cp). The rule *move_{ta}* chooses stochastically a *task*-node that does not yet belong to the underlying walk. The rules *cmpbest* and *free* are the same

⁸In the proposed solution, the *walk*-node and its outgoing edges are not deleted. The presence of these nodes and edges does not influence the correctness of the proposed solution, however it would be nicer to delete the useless walks, that is, all constructed walks up to the best one. This can easily be implemented, for example by using a kind specialized for cleaning the underlying graph after each iteration.

ant(GAP)

rules:



control: $init; ||init_{cp}||; (\set{move_{at}} \set{move_{ta}})^!; cmpbest; free$

Figure 5.10 The kind $\text{ant}(\text{GAP})$ which is specialized to construct walks for the generalized assignment problem

as in the kind *ant(TSP)* (see description above).⁹

5.6 Summary

In this chapter the major approaches to swarm computing have been surveyed and embedded into the framework graph-transformational swarms.

This chapter has modeled a discrete version of particle swarm optimization as a graph-transformational swarm. The main idea behind its discretization is the consideration that all information that specify the state of a particle are integers, each of which can be represented each by a node having a loop labeled by the corresponding value. Each particle is then presented by a node having outgoing edges to the components of its state. In this way all possible usual operations can be performed using graph transformation rules in a natural way. It was shown that the computations in the graph-transformational swarms version correspond with the runs in the corresponding discrete particle swarm.

Cellular automata has also been embedded into the graph-transformational swarms. The cells, their states and their neighbors, which together determine a configuration of a cellular automaton, are easily presented in a graph. Multidimensional cellular automata can be modeled using multidimensional sizes of graph-transformational swarms. Each cell is assigned a swarm member such that all members have the same kind specified by a graph transformation rule. Considering suitable representations of configurations as graphs, it was shown that cellular automata behave exactly like their swarm versions.

Ant colony optimization has been formulated as a framework composed of three components: (1) *ACO-input* which determines the elements needed as input for a specific ACO method, (2) ACO-procedures which is the fixed part in the framework describing the main procedures that a specific algorithm should apply and (3) ACO-parameters which contains amongst others the parameters that specify a given version in the ACO algorithms. Illustrative examples were provided for ACO-input and ACO-parameters. It was then shown how to formulate the different components in the graph-transformational swarms framework.

One of the aims behind embedding different swarm computing approaches into graph-transformational swarms was the development of new concepts that combine the ideas and advantages of these approaches. This chapter has provided implicitly an example where such a combination occurs. In the graph-transformational version of the ant colony optimization the members of kind *updater* are based on the ideas of cellular automata. They are not moving in the graph and perform computations based on the information

⁹The distribution functions f and g are specified in the scope of this work. See [66] for concert examples of f and g .

available in the neighboring nodes and edges (cells). The resulting swarm contains two kinds of members. Members that move (of kind *ant*) and others (of kind *updater*) that stay in their places and update the state of the whole environment by updating their own states in parallel. Using this combination of the concepts of cellular automata and ant colony optimization, it was possible to provide a version of ant colony optimization where parallelism is specified in a natural manner. The next chapter investigates in-depth the idea of a special case of members that are not moving in the graph called stationary members.

Chapter 6

Swarms with stationary members

This chapter studies a special type of members called *stationary members*. The stationary members are assigned to particular subgraphs of the considered environment graphs and stay there. They are responsible for calculations and transformations at the assigned areas. The number of such members is proportional to the size of the underlying graph if parts of the members' subgraphs are exclusively assigned. The advantage of stationary members is that it is easier to establish the applicability of rules and to guarantee that the members can act in parallel than for moving members.

Cloud computing for example is an engineering topic where swarms with stationary members can be applied in an adequate way, namely by modeling the nodes of the server network that form the cloud as stationary members. We illustrate the proposed concept by means of two case studies.

The current chapter is organized as follows: Section 6.1 sketches how cloud-based systems can benefit from the concept of stationary members. In Section 6.2 the notion of stationary members is defined. Section 6.3 demonstrates the notion of stationary members by means of two case studies. The chapter ends with a summary.

6.1 Cloud computing

Cloud computing is an emerging technology with high promises, but also with various technical challenges. A cloud consists of a network of computer systems that have different tasks providing resources as services. Such a network can be modeled in a natural way as a graph. Every computer system in the cloud can be represented by a node and the connections between them through labeled edges. The labels can encode various technical information. Given such a graph, the graph-transformational swarms with stationary members can be applied to solve problems and to analyze the behavior of the cloud.

The key connection of this chapter to cloud systems is the assignment of the stationary members to the nodes of the cloud network so that the members can execute calculations locally and parallel to each other. Depending on the problem and the architecture of the chosen network, it is possible to define different kinds of stationary members that have different roles. The technical details about how this can be achieved in a graph follow

in the rest of this chapter.

Graph-transformational swarms with stationary members can contribute to cloud-based engineering systems at least by (1) offering a visual and mathematical basis for the analysis of cloud behavior, and (2) providing a framework to design distributed algorithms based on parallel rule applications able to be used directly in a cloud. The first claim is directly inherited from the advantages using graph transformation as basic method in the proposed framework. The second claim is supported by the examples introduced in Section 6.3. The proposed solutions can be directly applied to a cloud system for detecting deadlocks.

6.2 Stationary members

The basic idea of swarm computation is that the members of a swarm can solve problems by team work and massive parallelism better or faster than a single processing unit. In the general setting, there may be two obstacles to meet these advantages. (1) To compute which member can perform which action may take time proportional to n^k where n is the size of the environment and k is the size of the left-hand side of the rule to be applied. The latter one can often be chosen small enough, but the former can get very large. (2) To make sure that members can act in parallel, one must check independence for each pair of matches of rules to be applied which is quadratic in the number of members in general.

Both obstacles can be avoided by stationary members. Their matches can be found locally in a small area rather than in the whole environment and independence of most pairs of matches is automatically guaranteed because they are always far away from each other. Although the environment is changed in a swarm computation, there may be an invariant part.

A member of a swarm may be considered as stationary if all left-hand sides of its rules can only match in the vicinity of a fixed subarea of the permanent part of the environment. Then the matches of the rules depend only on the size of this subarea and its vicinity and no longer on the size of the whole environment. Moreover, the independence of such a match from other matches must only be checked if the subareas and vicinities overlap which is never the case if the involved subareas are far enough away from each other. The following definition introduces the notion more formally,

Definition 30 (stationary members)

Let $S = (in, K, size, M, coop, goal)$ be a swarm. Then its members are called *stationary* if the following holds:

1. Each initial environment graph $G \in SEM(in)$ is associated with a set SUB of subgraphs which is kept invariant by swarm computations, i.e. SUB is a set of subgraphs of G' for each swarm computation $G \xRightarrow{*} G'$.
2. Each member $m \in M(k)$ for each $k \in K$ is associated with a subgraph $G(m) \in SUB$.

3. Each left-hand side of each rule of each member $m \in M(k)$ for each $k \in K$ contains a subgraph that matches only in $G(m)$ and the full match can be found in the neighborhood of $G(m)$.

Here neighborhood is a generic notion and may consist of all nodes adjacent to $G(m)$ and the edges incident to them or all nodes reachable by paths from $G(m)$ of a bounded length.

6.3 Examples

The notion of stationary members is illustrated by means of two simple and well-known decision problems.

6.3.1 Cycle freeness test

A swarm is provided with stationary members that tests an input graph for cycle freeness. Due to the massive parallelism of the swarm members' teamwork, the number of computational steps is linearly bounded. This example is kept simple. But to illustrate all the features of swarms, cycle freeness problem is solved in a dynamic setting, meaning that new edges are added to the underlying graph from time to time so that eventually every initial graph ends up with cycles.

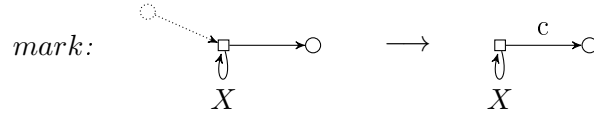
To keep the model as simple as possible, the swarm that tests a simple, unlabeled and directed graph G for cycle freeness receives this graph as a parameter where we assume, in addition, that the nodes are numbered from 1 to n . G is turned into the initial environment by adding an i -loop to each node $i \in [n]$. There are three kinds: (1) *marker* with a single rule *mark* which marks an edge outgoing of a node with an X -loop, provided that there is no incoming unlabeled edge. The control condition $||mark||$ requires that the rule be applied with maximum parallelism. The size is the number n of nodes. The member $marker_i$ for $i \in [n]$ is obtained from *marker* by relabeling all occurring X with i . (2) *resetter* has just a single rule that turns a marked edge into an unmarked one. The X - and Y -loop identify the source and the target yielding stationarity of the members $resetter_{i,j}$ where X and Y are relabeled by the node identifier $i, j \in [n]$. (3) *adder* has a rule that adds an edge between an X - and a Y -looped node. The control condition $[add]$ requires that the rule may be applied or not. The cooperation condition requires that *marker* is applied as long as possible, followed by an arbitrary number of repetitions of *resetter* followed by *adder* followed by *marker* repeatedly as long as possible. The goal is to reach a graph without unlabeled edges meaning that all unlabeled edges are changed into c -marked ones. The swarm and its kinds are schematically specified in Figure 6.1.

As the rule applications of the swarm members of kind *marker* change unlabeled edges into c -marked ones, the nodes and their loops are kept invariant so that the members are stationary and the match of the rule *mark* of the member $marker_i$ is fixed by the unique i -loop and varies only in the outgoing edges. Two matches of *mark* of the

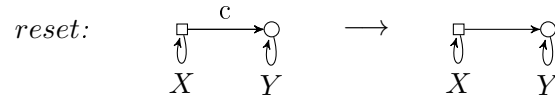
cyclefree(G)
 initial: $id\text{-looped}(G)$
 kinds : $marker, resetter, adder$
 size : n, n^2-n, n^2-n
 members: $marker_i$ for $i \in [n]$
 $resetter_{i,j}$ for $i, j \in [n], i \neq j$,
 $adder_{i,j}$ for $i, j \in [n], i \neq j$
 coop: $marker!$
 goal: $forbidden(\bigcirc \longrightarrow \bigcirc)$

marker

rules:

control: $\|mark\|$ **resetter**

rules:

control: $reset$ **adder**

rules:

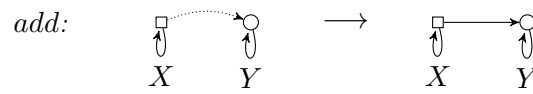
control: $[add]$

Figure 6.1 The *cyclefree* swarm with its kinds *marker*, *resetter* and *adder*

member $marker_i$ are independent if they access different outgoing edges which can be checked locally. Matches of $mark$ that are preformed by two different members are always independent. Consequently, a maximum parallel step of the swarm marks simultaneously all edges outgoing from nodes without incoming unlabeled edges. As long as there are such edges, their number decreases in each computational step of the $marker$ -members such that - by induction - we always end up with the unlabeled edges on cycles. In particular, the number of such steps is bounded by the length of the longest simple path and cycles are never broken. Summarizing, the following result is shown.

Theorem 4

The swarm $cyclefree(G)$ reaches its goal if and only if G is cycle-free. To decide this, the number of steps is bounded by the length of the longest simple path in G .

Moreover, the $resetter$ -step returns the graph given before the $marker$ -computations, and the $adder$ -step adds new edges. The resulting graphs are tested by the $marker$ -computations for cyclefreeness in the same way as the initial graph. Therefore, Theorem 1 holds for them, too. Note that $resetter_{i,j}$ and $adder_{i,j}$ are stationary members fixed by the nodes i and j and the corresponding loops.

6.3.2 Reachability test

In analogy to cycle freeness example, the swarm takes as parameter a simple, unlabeled and directed graph G , where the nodes are numbered from 1 to n . In order to test if a node t is reachable from a node s the swarm has additionally, two parameters $s, t \in [n]$. The graph G becomes an initial environment by adding an i -loop to each node $i \in [n]$. The swarm has a unique kind $catcher$ with a single rule $catch$. This rule creates an s -loop at a node with no s -loop, if there is an edge from an s -looped neighbor to this node. According to the control condition, this rule is applied if possible. Every node beside s gets a member. The members $catcher_i$ for $i \in [n] - \{s\}$ is obtained by relabeling every occurrence of X with i . The cooperation is *free* and the goal is to spread the s -loop to reach the node t . The members are stationary because the nodes and their loops are kept unchanged. Figure 6.2 specifies the swarm $reachable(G, s, t)$ and the kind $catcher$.

One can imagine that in every node, beside s , a member $catcher_i$ is sitting and waiting (sleeping) until an incoming neighbor gets an s -loop. If this happen, then $catcher_i$ "catches" the s -loop by creating a copy in the underlying node. After that, $catcher_i$ sleeps again. In the next step all outgoing neighbors of $catcher_i$ catch the s -loop.

The matches of two rules $catch$ from different members are always independent. Accordingly, a maximum parallel step of the swarm spreads simultaneously an s -loop to all nodes without an s -loop that are outgoing neighbors of nodes with s -loops. If t is reachable from s , then the number of such steps is equal to the length of the shortest simple path from s to t . If not, then the number of steps is bounded by the length of the longest simple path starting from s .

Theorem 5

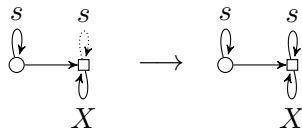
The swarm $reachable(G, s, t)$ meets its goal if and only if the node t is reachable from

```

reachable( $G, s, t$ )
  initial:  $id\text{-looped}(G)$ 
  kinds :  $catcher$ 
  size :  $n-1$ 
  members:  $catcher_i$ , for  $i \in [n]-\{s\}$ 
  coop:  $free$ 
  goal:  $required(\overset{s}{\circlearrowleft} \circlearrowright \overset{t}{\circlearrowleft} \circlearrowright)$ 

```

```

catcher
  rules:
    catch : 
    control:  $try(catch)$ 

```

Figure 6.2 The swarm *reachable* with its kind *catcher*.

the node s in G . The number of steps to decide this is bounded by the minimum of the length of the shortest path from s to t and the length of the longest simple path starting from the s node.

6.4 Summary

In this chapter, the notion of graph-transformational swarms with stationary members is introduced. Stationary members are assigned to particular subgraphs of the considered environment graphs and are responsible for the local calculations and transformations. Their advantage is that it is easier to establish the applicability of rules and to guarantee that the members can act in parallel compared to the general case where the members are moving. This chapter proposes cloud computing as an application field for the notion of graph transformational swarms with stationary members. The key is the modeling of a cloud as a graph. Two case studies that are directly related to cloud computing were presented. The case studies show that such solutions take advantage of massive parallelism, can be visually represented and support correctness results. However, in order to prove the power of the concept, bigger and more difficult examples should be modeled in the future. In cloud computing specially, one can consider task scheduling problems which are NP- hard problems in general and can profit from a combination of swarm heuristics and the massive parallelism in the proposed framework.

Chapter 7

Modeling in dynamic logistic networks

As logistic networks get larger and larger and more and more complex, they become more difficult to handle and to control. The traditional central control does not work flexibly and efficiently enough in any case so that one must look for alternative approaches. This applies particularly if the logistic network may change dynamically. One of the most significant current paradigms that faces this complexity is the so-called autonomous control approach (cf. [46]). This approach proposes that each logistic object, such as a container or an automated guided vehicle, receives its own computing processor and makes its decision autonomously. Therefore, the components can react locally and quickly to changes in the environment. However, a major challenge within this kind of decentralized approach is how the individuals act and cooperate with each other to reach a desired global goal. This applies and discusses graph-transformational swarms as a formal modeling approach to dynamic logistic networks with decentralized control. As an illustrative example, the routing problem of automated guided vehicles is considered and discussed.

This chapter is organized as follows. Section 7.1 discusses how graph-transformational swarms can be used to model dynamic logistic networks. To illustrate this, essential aspects of the routing problem of automated guided vehicles are modeled as graph-transformational swarms in Section 7.2. Section 7.3 summarizes the chapter.

7.1 From swarms in nature to logistic networks

This section argues that graph-transformational swarms as introduced in the previous chapters are appropriate means to model dynamic logistic networks. Several approaches to swarm computation, including graph-transformational swarms, mimic swarms in nature, as pointed out in Chapter 2 in more detail. The interesting aspect is that already, swarms in nature solve problems closely related to logistics. Moreover and more interestingly in the context of this chapter, a closer look the other way round at

dynamic logistic networks in Subsection 7.1.2 reveals that they can be considered as graph-transformational swarms.

7.1.1 Relating swarms in nature with logistics

The proposed framework is inspired by the swarm behavior in nature which describes the group behavior of social animals. Chapter 2 has shown that several studies agree on the assumption that the swarm behavior results from relatively simple rules on the individual level (see, e.g., [9, 16, 18, 74]). In biology the underlying phenomena is also known as *self-organization*: The individuals in the group interact locally with other group members and have no knowledge of the global behavior of the entire group. Furthermore, all members play the same role without any hierarchical structure [9].

Chapter 2 has also shown that, based on swarm behavior, social animals solve continuously complex problems. Ant colonies as well as bee hives construct nests and manage the resources inside of them. They forage for food and transport it in an efficient and flexible way, based on the cooperation with each other and the communication through the environment. Obviously, such phenomena have logistic aspects. Therefore, it is somewhat evident that the behavior of swarms in nature has inspired the introduction of such concepts as artificial intelligence and swarm computation that are based on the idea of self-organization to solve logistic problems. One encounters some approaches to swarm computation in the literature (see, e.g., [6, 7, 33, 50, 71]) where logistic problems are solved as typical examples, such as the shortest-path problem, the traveling-salespersons problem and others. One may summarize that the passage from swarms to logistics is not very long.

7.1.2 Dynamic logistic networks as graph-transformational swarms

On the other hand, consider dynamic logistic networks. Their underlying structures consist of nodes and connecting edges. The nodes represent logistic hubs of different types like production sites, storage facilities, car pools, etc. or - on a more detailed level - packages, containers, cars, trucks, etc., and the edges represent transport lines or information channels or the like. Without loss of generality, one can assume that there is always some start structure. To manage the flows of material and information in a logistic network, various logistic processes are running. If the network is large and widely distributed, then it may not be meaningful to control the processes centrally. Alternatively, the logistic processes in the network may run simultaneously and independently of each other, each performing its own actions and following its own autonomous control. But such a decentralized control requires coordination and cooperation whenever material or information must be exchanged carrying out the overall tasks. To coordinate autonomous logistic processes in a network in such a way that the cooperation works properly, becomes even more difficult if the network changes structure dynamically. One needs appropriate modeling methods like those provided by graph-transformational swarms.

The underlying structures of dynamic logistic networks are defined as graphs so that

dynamic logistic network	graph-transformational swarm
underlying structure	environment graph
types of logistic entities	kinds
logistic entities	members
possible actions	rules
autonomous control	control conditions
start structures	initial environments
coordination	cooperation conditions
tasks	goals
simultaneous and decentralized processing	massively parallel rule application

Table 7.1 Correspondence between dynamic logistic networks and graph-transformational swarms

they correspond directly to environment graphs of graph-transformational swarms where the initial environments play the role of the start structures. The various types of logistic entities like hubs, sites, carriers, containers, etc. together with the actions that are performed on them or affect them can be seen as kinds so that the entities themselves are the swarm members. In particular, the possible process actions correspond to the rules and the autonomous control is reflected by the control conditions. Finally, the coordination of the processes running on the logistic networks is embodied by the cooperation conditions and the overall tasks by the goals.

Summarized in Table 7.1, there is a very close relationship between the main features of dynamic logistic networks and the syntactic components of graph-transformational swarms. Moreover and most interestingly, the idea of autonomous processes that run simultaneously and decentralized in a logistic network is well reflected on the semantic level of graph-transformational swarms, as all the members always act in parallel.

7.1.3 The potentials of the approach

We propose in this chapter to model dynamic logistic networks by means of graph-transformational swarms. In the previous subsection, one can see that the notion of such swarms covers all the main features one expects and finds in dynamic logistic networks. Nevertheless, one may wonder which particular potentials and advantages this approach provides:

1. The conception of graph-transformational swarms offers a formal framework with a precise mathematical semantics based on massive parallelism of rule applications.
2. As the environments are graphs and the processing is modeled by graph transformation rules specified by four graphs each, the approach provides a fundament for visualization so that it can be considered as a visual modeling approach.

3. In fact, graph-transformational swarms can be executed on graph transformation engines like GrGen.NET (see [37]) or AGG (see [96]) so that visual simulation is possible for illustrations, tests and experiments of various kinds. The next section for example illustrates in Figures 7.3 and 7.7 computations generated by GrGen.NET. They visualize the computational steps in a simple environment in order to make it easier for a reader to understand how the developed swarm behaves. Moreover, the implementation in GrGen.NET allows us a visual testing using different graphs.
4. The formal semantics is based on derivations which are sequences of rule applications. Therefore, a proof technique is provided by induction on the lengths of derivations.
5. If one fixes the initial environment and bounds the lengths of derivations, then the behavior of graph-transformational swarms can be translated into formulas of the propositional calculus so that SAT-solvers can be employed for automatic proving of properties, as far as they are expressible in propositional calculus (cf. [34]). A typical correctness property one would like to prove in this way is: Will the goal be reached? Another property of interest that can be proven in this way is deadlock freeness.
6. The approach is very flexible and generic because all the modeling concepts can be chosen from a variety of possibilities. This applies to the kind of graphs which may be directed or undirected, labeled or unlabeled, connected, simple, etc. It applies similarly to the kind of rules, control conditions and graph class expressions. The actual choice may depend on the application at hand or the taste of the network designers.
7. Graph-transformational swarms do not need extra features to make logistic networks dynamic, i.e. to allow the modeling of dynamic changes of the underlying structure. The members of the swarm perform their tasks by applying rules to the environment graph. This includes the possibility of members to change the environment structurally.

7.2 Routing of AGVs by a graph-transformational swarm

This section proposes a solution to the routing problem of the automated guided vehicles (AGVs) using the notion of graph-transformational swarms. Automated guided vehicles are driverless transportation engines that traditionally follow guide paths like lines on the ground. Their use has expanded rapidly in the last decades. Beside the classical application in small manufacturing systems, nowadays, the tendency is to use AGVs more and more for transport in highly complex systems including external areas like container terminals (for a general overview, see, e.g., [65, 100]). One of the important problems that a designer of an AGV system faces in complex areas is the collision-free

routing problem. The classical way to solve this problem is the central time windows planning (see, e.g., [93, 97, 98]). However, the tendency in the last years is to explore more decentralized approaches (e.g., [92, 103]). In the same vein, this section proposes a decentralized solution using the notion of graph-transformational swarms.

7.2.1 The routing swarm

The infrastructure where the AGVs operate is modeled as an *id-looped distance* graph. In a graphical representation the nodes correspond to the ends or intersections of paths including important stations like pick-up and delivery locations. The edges represent the paths or segments of paths in the infra-structure depending on their lengths. The distance of an edge can correspond to the distance of the corresponding path or to some cost of traversing it.

This thesis proposes a solution based on two stages. The first one consists of the preparation of the layout in a such way that the AGVs follow only local information later. The second one consists of the navigation process of the AGVs depending on an arbitrary task assignment.

```

routing(m)
  initial:  id-looped(distance)
  kinds :   preparator,resolver,assigner,navigator
  size :   n = #nodes,n,m,m
  members: preparatori for i ∈ [n]
               resolverj for j ∈ [n]
               assignerk for k ∈ [m]
               navigatorl for l ∈ [m]
  coop:    preparator; (assigner; (resolver; navigator)*)*
  goal:    all vehicles arrived

```

Figure 7.1 The schematic representation of the graph-transformational swarm *routing*

The parameter m is the number of AGVs and can be chosen freely. The swarm has four kinds: *preparator*, *resolver*, *assigner*, and *navigator*. Their sizes are n, n, m and m respectively where n is the number of nodes in the underlying graph $G \in SEM(id-looped(distance))$. The members are obtained by relabeling in such a way that every node in the graph gets assigned two member one of kind *preparator* and the other of kind *resolver*, and similarly, every AGV gets assigned a member of kind *assigner* and a member of kind *navigator*. How relabeling is achieved is described below in the detailed introduction of the kinds.

The cooperation condition requires that *preparator* is applied realizing the layout preparation followed by an arbitrary repetition of assignments, each followed by an arbitrary number of conflict resolving and navigation. The goal is that all AGVs reach their

assigned targets. The swarm is schematically presented in Figure 7.1.

As mentioned before, we have implemented the swarm *routing* in the graph transformational tool GrGen.NET. The resulting computation steps of an experiment with an environment composed of a very small graph and three AGVs are used in this section to accompany the explanation of the behavior of the swarm *routing*. Figure 7.3 summarizes the layout preparation process and Figure 7.7 illustrates the remainder of the computation which consists of the assignment and the conflict free navigation of the AGVs.

7.2.2 Layout preparation

The layout preparation equips the underlying graph with additional edges in such a way that every node in the graph can indicate an AGV having the target T whose next node can be visited to reach T with the minimal distance possible. Given an i -node, let us code such an *indicator* as an outgoing edge e labeled with a pair T, D . Let us say that i has an indicator to T with the distance D using the successor s , where s is the destination of e . If D is minimal considering simple paths up to the maximal lengths l , we say that the indicator is called l -minimal. If D is minimal considering all possible paths, then the indicator is optimal. A path composed of indicators to a target T is called an indicator path to T . If every node in the graph has only optimal indicators to every reachable node, then the graph is called *fully indicated*.

preparator

rules:

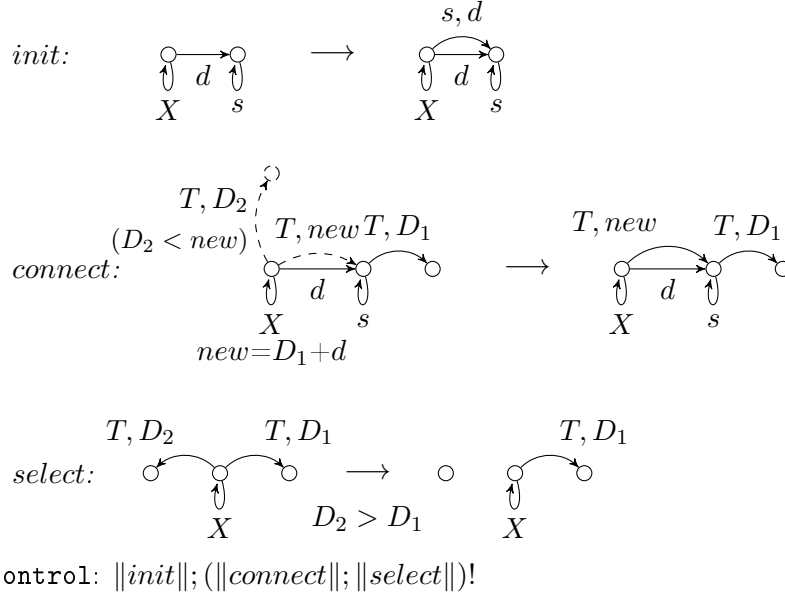


Figure 7.2 The unit *preparator*

The members of kind *preparator* realize the layout preparation process. The kind *preparator* specified in Figure 7.2 initializes this process with rule *init*. It adds an indicator in an X -node to a direct successor s provided that such an indicator does not yet exist. The rule *connect* connects an X -node with an existing indicator path to T . It is applied if a direct successor s of X exists, having an indicator to T with a distance D_1 , provided that there is no other direct successor of X having the same target T with a distance D_2 such that $D_2 < D_1 + d$. The rule *connect* generates an indicator to T with the new distance $D_1 + d$ using the successor s . If an X -node has two indicators to a target T with different distances, the rule *select* deletes the one with the larger distance, selecting in this way the best one to be kept. The control condition requires that the rule *init* is applied with maximum parallelism. Afterwards, the rule *connect* is applied; followed by *select*, both with maximum parallelism. Because of the negative application constraint of *init*, for every successor node *init* is applied only once in the whole swarm computation while *connect* and *select* are iterated as long as possible according to the control condition of *preparator*.

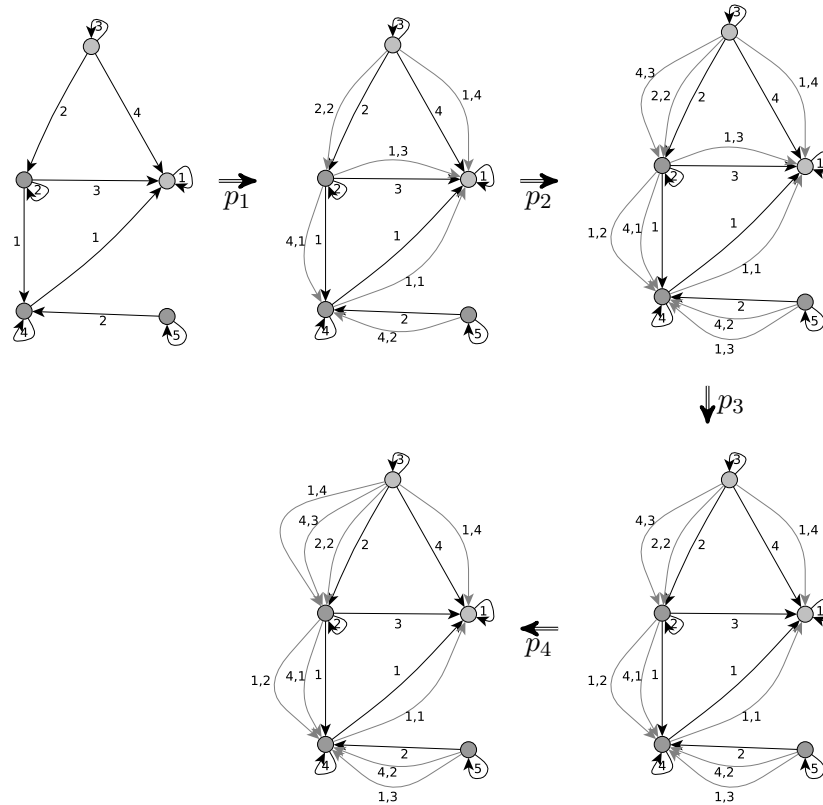
In the swarm, there are n members of kind *preparator*. The member *preparator_i* for $i \in [n]$ is obtained from *preparator* by relabeling all occurring X with i . Then i becomes a fixed label in *preparator_i*. The role of the other labels must be explained now: They are placeholders for all possible values so that the rules are rather rule schemata that must be instantiated before they are applied. A control condition like $||init||$ means accordingly, that the maximum number of the instantiations of the rule *init* must be applied in parallel. This mechanism that keeps the representation of rule sets small is used in all our examples of transformation units.

The following describes how the members work together using the computation in Figure 7.3 as illustrating example. In the first step, all members apply the rule *init* in parallel, generating in every node indicators to all successor nodes (see the result of the derivation p_1 in the example). In the second step, the parallel application of the rule *connect* in parallel connects all nodes to construct indicator paths of length 2. It also connects those that are already connected to indicator path of length 1 if the new distance is smaller than the old one (in the example, p_2 adds indicators in the nodes 2 and 3). In the third step, all members apply *select* in parallel deleting all indicators using paths of length 2 and 1 that are not 2-minimal (p_3 deletes the indicator in 2 to 1 with distance 3 keeping the minimal indicator to 1 with distance 2). Note that a node can have more than one minimal indicator to the same target (in the example, the node 3 is given two indicators to 1 with the same distance 4). By induction, one can prove that in $2L - 1$ steps all L -minimal indicators are constructed. If the longest path with a minimal distance is constructed, then the *preparator*-members cannot apply any rule anymore (except the sleeping rule) and the constructed indicators are optimal. Because the length of such a path is shorter or equal $n - 1$, the number of steps is bounded by $2n - 3$. In summary, the following correctness result holds:

Theorem 6

Given an id-looped distance graph G , the swarm *routing* transforms it by the initial

Note that the layout preparation process can be considered as a distributed version of the Dijkstra’s shortest path algorithm (cf. [19]).



The behavior of the swarm in the layout preparation stage can also be interpreted as follows: In the first step, a change in the environment is introduced using the rules *init*. The swarm reacts by propagating this information backwards over all nodes, combining the rules *connect* and *select* of all members of kind *preparator*. In more sophisticated versions of the underlying swarm, one can consider that additional changes can occur in the environment. For example, the suppression of indicator edges can simulate a traffic congestion. Such a change can also be handled in the same way by propagating the information backward to all concerned members. For illustration purposes, the preparation process here is kept simple and the next subsection shows how the automated guided vehicles can use the generated information to navigate to their assigned targets conflict-free.

7.2.3 Assignment, conflicts resolving and navigation

The kinds *assigner* and *navigator* model the task assignment and navigation process from the point of view of the AGVs. However, the task assignment has the most simple form, serving solely the simulation purposes of the computational steps. The kind *resolver* models the conflict resolving from the point of view of a node that multiple AGVs have it as next destination and would like to visit the same next position, which is determined by a direct successor of the underlying node.

An AGV is encoded as an AGV-edge labeled by a, T, p where a is the name of the AGV, $T \in [n]$ corresponds to its assigned target and $p \in \mathbb{N}$ is its current priority. A vector of nodes $\langle n_1, n_2 \rangle$, such that an indicator (T, d) from n_1 to n_2 exists, is called therefore an AGV position. It is considered that AGVs with target T can occupy such a position with the restriction that at most one AGV can be present in a position at a given time. The priority is needed to resolve conflicts if more than one AGV compete for the same position.

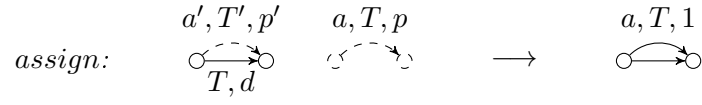
The kind *assigner* has just a single rule *assign* that creates a vehicle edge labeled with a, T, p between two arbitrary nodes, provided that this position leads to the target T and is free and that the vehicle edge is not yet present in the whole graph. Note that the edges labeled by T, d in the rule *assign* have different forms meaning that they do not belong to the gluing graph of the rule. The control condition $[assign]$ requires that the rule may be applied or not, so that not every vehicle must be present any time. The members $assigner_j$ for $j \in [m]$ are obtained from the kind *assigner* by relabeling all occurring a with a_j and the a' by a_k for $j \neq k$.

The kind *resolver* has a single rule *reserve*. It reserves for an incoming AGV-edge labeled by a, T, p a next possible position $\langle X, s \rangle$ having an indicator T, d provided that the following four negative contexts are all satisfied: (1) There is no other concurrent AGV (represented in the rule by the incoming AGV-edge a_1, T_1, p_1) with a higher priority ($p_1 > p$), and can visit too the position $\langle X, s \rangle$ (see the edge (T_1, d_1) in the rule). This negative context with two edges is bordered by a dotted line to indicate that the two parts should be satisfied together. (2) The position $\langle X, s \rangle$ is free: there is no other outgoing AGV-edge labeled by a_2, T_2, p_2 parallel to the (T, d) -edge. (3) There is no reservation a_3 for any other vehicle in the next position $\langle X, s \rangle$. (4) The AGV a has not yet a reservation: there is no outgoing edge labeled by a . The rule *reserve* adds an outgoing edge labeled by a parallel to the (T, d) -edge which indicates that the underlying position is reserved for the AGV a . The rule *reserve* may be applicable for two AGVs with the same priority both claiming the same next possible position $\langle X, s \rangle$. But the control condition requires that the rule is applied sequentially as long as possible so that only one of the potential reservations is chosen non-deterministically. The member $resolver_j$ for $j \in [n]$ is derived from *resolver* by relabeling all occurring X with j . This means in particular that reservations are done sequentially at the node with the j -loop, but in parallel for different nodes.

The kind *navigator* contains three rules *wait*, *move* and *arrive*. The rule *wait* increments the priority p with 1. The rule *move* is responsible for the forward movement of the AGV until the target is reached. It moves the AGV a with the target T forward

assigner

rules:

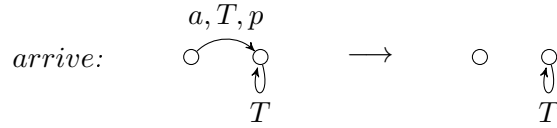
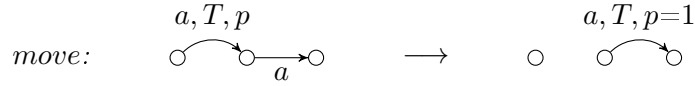
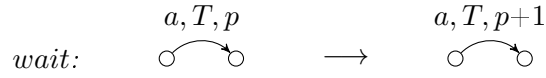


control: [assign]

Figure 7.4 The unit *assigner*

navigator

rules:

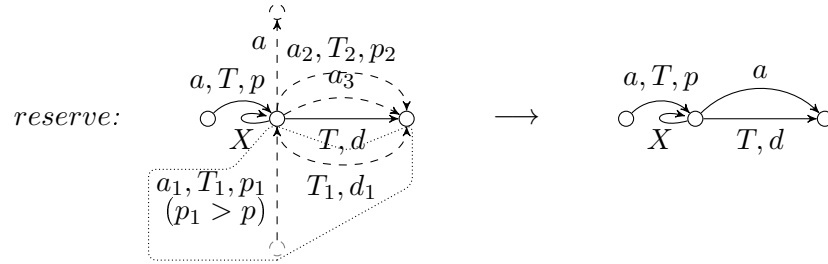


control: *move|arrive|wait* & *wait* < *move* & *wait* < *arrive*

Figure 7.5 The unit *navigator*

resolver

rules:



control: *reserve*!

Figure 7.6 The unit *resolver*

following an a -edge (which is added by a *resolver* member). If the target node is reached, the rule *arrive* can be applied. The rule *arrive* deletes the AGV-edge, signaling in this way to the task assigner that the AGV a is free for a new assignment. The control condition requires that one of the rules *move*, *arrive* or *wait* is applied. Therein, *wait* has the lowest priority. The member $navigator_k$ for $k \in [m]$ are obtained from *navigator* by relabeling all occurring a with a_k .

After the layout preparation, only members of kind *assigner*, *resolver* and *navigator* are active. The *assigner* members create an arbitrary number less or equal m of AGV-edges in parallel. According to the parallelization theorem together with the fact that the T, d edges in the rule *assign* do not belong to the gluing graph, the positions of the created AGVs are pairwise different ensuring a conflict-free assignment. Afterwards, all created AGV-edges act in parallel by moving forward or waiting depending on the decision of the *resolver* members which are present in every node to check for and to resolve conflicts. They reserve the next possible position of the AGVs based on their priorities. The AGVs with a reservation are moved forward, setting their priorities to one, the others that arrive at their targets have their corresponding edge deleted, all others have to wait, incrementing their priorities by one. If the number of repetitions is high enough, the swarm reaches its goal, otherwise the process starts again by assigning new tasks to inactive vehicles. The swarm repeats this process until the goal is reached. Especially, the following result is obtained.

Theorem 7

If the swarm *routing* reaches its goal, each AGV that has been assigned to a target reaches this target collision-free.

Proof Consider a computation $G_0 \xRightarrow{*} G_n$ of the swarm *routing*. According to the cooperation condition, an initial section $G_0 \xRightarrow{*} G_i$ for some i prepares the initial environment G_0 into a graph with the properties stated in Theorem 1. And the tail $G_i \xRightarrow{*} G_n$ is composed from sections of the form $G_{k_j} \xRightarrow{*} G_{k_j+1} \xRightarrow{*} G_{k_{j+1}}$ for $i = k_1 < \dots < k_m = n$, $m \geq 1$ where, for $j = 1, \dots, m$, the first step is an *assign*-step and the remainder repeats *resolver*-steps followed by *navigator*-steps. For $m = 0$, this is the empty sequence. Then the theorem can be proved by induction on m . For $m = 0$, no car moves so that no collision can happen. Let now the computation have $m+1$ *assign*-steps. Due to the induction hypothesis, the vehicles run collision-free for the first m *assign*-steps. The $(m+1)$ -st *assign*-step adds some further AGVs, but only if none of these is already present and the edges where the vehicles are assigned are not occupied. All further steps are applications of the rule *reserve* alternated with the applications of the rules *wait*, *move*, and *arrive*. A collision would only happen, whenever two AGVs move onto the same edge at the same time. But such collision is impossible because the entered edge is reserved before by exactly one vehicle as discussed in detail above in the explanations of the kinds.

The swarm *routing* is designed to solve conflict-free situations where two or more concurrent AGVs want to traverse the same node. However, it should be mentioned that

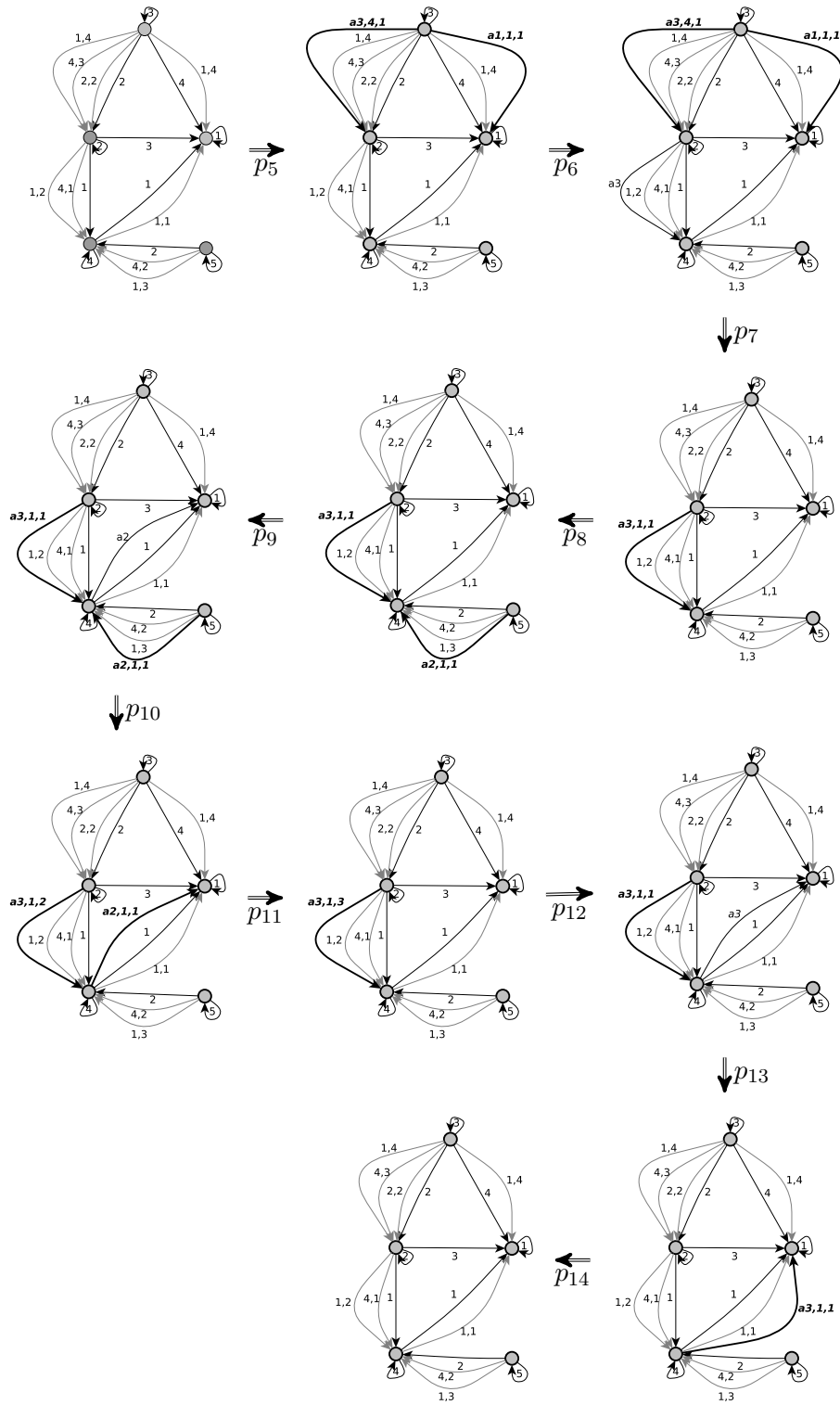


Figure 7.7 A sample computation of the swarm *routing*, illustrating the navigation process

the presented swarm does not handle deadlocks caused from circular waits. The characterization, detection and avoidance of such situations should be treated in future work. Figure 7.7 illustrates the computations in the navigation process of three AGVs a_1, a_2 and a_3 starting with the fully connected graph resulting from the preparation process in Figure 7.3. In the first step of this stage, two AGVs are arbitrarily chosen to receive assignments and the remainder is kept inactive. a_1 and a_3 are assigned the targets 1 and 4 and the start positions $\langle 3, 1 \rangle$ and $\langle 3, 2 \rangle$ respectively as a result of the application of the rule $p_5 = assign_1 + assign_3 + sleep_2$, where the indices in the rules correspond to the indices of the members that apply them. After the assignment, the member $resolver_2$, which resolves conflict in node 2, reserves the next position for the AGV a_3 by generating an a_3 -edge while all other members of kind *resolver* apply their sleeping rule i.e., $p_6 = reserve_2 + \sum_{i \in [5] \setminus \{2\}} sleep_i$. The rule $p_7 = arrive_1 + move_3 + sleep_2$ is applied, making a_1 available for other assignments because it has already arrived at its target, and moving forward the vehicle a_3 to its reserved position. The AGV a_2 is still sleeping in this step. At this point, the repetition of resolving and navigation is finished, allowing that the assignment starts again. $p_8 = assigner_2 + \sum_{i \in [3] \setminus \{2\}} sleep_i$ assigns the target 1 and the start position $\langle 5, 4 \rangle$ to the AGV a_2 . The other members of kind *assigner* sleep because the a_1 is not chosen to be assigned and a_3 is already assigned. In the next step, the rule $p_9 = resolver_4 + \sum_{i \in [5] \setminus \{4\}} sleep_i$ chooses a_2 to move forward, reserving the only possible next edge for it. This step illustrates the behavior of a member resolver in a situation where concurrent AGVs having the same maximal priority and want to traverse the assigned node. Namely one is chosen arbitrarily. Following this decision, a_2 moves forward and a_3 waits, augmenting its priority by 1 as a result of the rule $p_{10} = move_2 + wait_3 + sleep_1$. The navigation rules yield that a_2 applies its *arrive* rule and that a_3 waits again because the next position is occupied, that is $p_{11} = arrive_2 + wait_3 + sleep_1$. In the next two steps $p_{12} = resolver_4 + \sum_{i \in [5] \setminus \{4\}} sleep_i$ reserves the next position for a_3 followed by $p_{13} = move_3 + \sum_{i \in [2]} sleep_i$ which moves it to the reserved position. In the last step, and because the last active AGV a_3 arrives at its target $p_{14} = arrive_3 + \sum_{i \in [2]} sleep_i$, the swarm reaches its goal.

7.3 Summary

This chapter proposed modeling dynamic logistic networks with decentralized processing and control by means of graph-transformational swarms. The members of such a swarm act and interact in a common environment graph. It is a rule-based approach, the semantics of which is based on massive parallelism according to local control conditions of the members and a global cooperation condition of the swarm as a whole. As was discussed above, this corresponds to dynamic logistic networks with their logistic hubs and their processes which run simultaneously and autonomously with a proper way of coordination. It was sketched out how automated guided vehicles and their routing can be modeled by a graph-transformational swarm as an illustrative example. In this example, an innovative decentralized control has been proposed. The main idea behind it is to equip the environment with computing devices, that perform several computations, relieving the AGVs from calculating the whole path between a start and a target position.

The environment can act to changes like by propagating such information based on neighboring communication. The AGV use only local information provided from the environment. This example has demonstrated the capability of the approach regarding visualization in the design level as well as the computation level. Furthermore, two theorems have been provided using the advantage of the formal semantics of graph-transformational swarms.

Chapter 8

Conclusion

8.1 Summary

In this thesis, the concept of graph-transformational swarms was introduced and studied. The concept combines the ideas of swarms and swarm computing and the formalism capability of graph transformation.

The major approaches of swarm computing as well as their biological foundations are surveyed using a unifying perspective. That is, focusing on the common principles, with the aim to propose a concept that embeds the different swarm computing concepts. The identified ideas and principles are formulated by means of graph transformation. For this purpose, a suitable graph transformation approach is chosen, adapted and extended.

The resulting concept consists of a system composed of a set of members that act and interact in an environment represented by a graph. The members are structured into kinds such that each kind specifies the role to play in the swarm. The number of members of a kind is given by the size of the kind. Using the notion of vector size, it is possible to define multidimensional swarms where the members can be assigned to points in a multidimensional space. Furthermore, unbounded sizes are allowed. A kind corresponds to a simple graph transformation unit which consists of a set of rules and a control condition that regulate the rule applications. The members of a kind are modeled as units related to the unit of this kind, assuring in this way that all members of some kind are alike.

A swarm computation starts with an initial environment and consists of iterated rule applications. In every step, each member of the swarm applies one of its rules, requiring therefore massive parallelism. The choice of rules depends on their applicability and the control condition of the members. The applicability of rules is well specified by the individual conditions of each candidate rule and by means of the generalized parallelization theorem that specifies the conditions for parallel application of rules of different members. Another possibility to regulate the rule applications is the cooperation condition. It regulates the alternation between different kinds, determining therefore which members are allowed to participate in a computation step. Finally,

a swarm may have a goal also given by a graph class expression. A computation is considered to be successful, if an environment that meets the goal is reached.

The concept of graph-transformational swarms provides a formal framework for the study of swarm computation. In many swarm approaches, the environments of the swarms are either chosen as graphs explicitly or can easily be represented by graphs. And because rules are widely and successfully used as the core of computation, graph transformation combining rules and graphs is a natural candidate for the formalization of swarm computation. This is shown by embedding the major swarm computing methods into the graph transformation framework.

This thesis presents several examples where graph-transformational swarms is successfully employed to solve problems. First, simple illustrative examples are used, followed by classical optimization problems, and finally a sophisticated logistic problem is modeled. Therein, it was possible to demonstrate the capability of the approach regarding visualization on the design level as well as on the computation level. Furthermore, correctness theorems are formulated and proved using the advantage of the formal semantics of the concept. In summary, this thesis provides a framework that offers the following features and advantages:

- Graphs and rules are mathematically well-understood and quite intuitive syntactic means to model algorithmic processes. Moreover, the additional use of control and cooperation conditions as well as graph-class expressions allows very flexible forms of regulation.
- Derivations as sequences of rule applications provide an operational semantics that is precise and reflects the computational intentions in a proper way.
- Based on the formally defined derivation steps and the lengths of derivations, the approach provides a proof-by-induction principle that allows one to prove properties of swarm computations like termination, correctness, efficiency, etc.
- In the area of graph transformation, one encounters several tools for the simulation, model checking and SAT-solving of graph transformation systems that can be adapted to graph-transformational swarms.
- And maybe most important, the proposed framework offers systematic and reliable handling of massive parallelism. In several swarm approaches, the simultaneous actions of swarm members are organized in a very simplistic way by avoiding any kind of conflict or are required, but not always guaranteed (cf. e.g., [75]). In contrast to that, the simultaneous actions of members of graph-transformational swarms are assured whenever the member rules are applicable and pairwise independent. Both can be checked locally and much more efficiently than the applicability of the corresponding parallel rule.

8.2 Contributions

This thesis makes several noteworthy contributions to the current state-of-art in science and engineering. The scientific results are embedded in a framework that offers a base for future studies and exploration of swarm computing and distributed networks. In more detail, the following contributions are obtained:

- The general ideas and principles of swarms and swarm computing are identified. The formulation is based on theories and findings from biology and studies of the major swarm computing methods.
- The generalized parallelization theorem is introduced and proved. It proposes a generalization to the well-known parallelization theorem considering an arbitrary set of direct derivations rather than only two direct derivations. The generalized parallelization theorem offers a theoretical basis but also practical techniques to specify and handle massive parallelism.
- The research on graph transformation units is complemented and enriched. Graph transformation units are adapted and successfully used as the basic modeling components for kinds and members in swarms. Their modeling power has been, therefore, confirmed. Furthermore, the use of relabeling to generate similar units is an intuitive technique that is proposed here, as alternative to the classical inheritance methods.
- Stochastic control is introduced as a new control condition. It extends the concept of control condition to specify matching based on distribution functions. Therefore, it offers the possibility to model stochastic processes not only in the field of swarm computing but also in other fields of soft computing.
- The new notion of stationary members is introduced. The advantage of the notion is its efficiency regarding the examination of the applicability of rules. Cloud computing is proposed as an application field for the notion of graph transformational swarms with stationary members. However the notion can be used as modeling approach in many other fields where computing devices are distributed.
- A distributed version of the well-known ant colony algorithm *MZN-MAX* ant system is proposed. It is a natural result of the formulation using graph-transformational swarms. Besides the ants, other stationary members are assigned to edges and are responsible of updating of the corresponding pheromone locally.
- In this thesis, it is argued that graph-transformational swarms are suitable for modeling dynamic logistic networks with decentralized processing and control in general. In particular, an innovative decentralized control that solves the routing problem of automated guided vehicles is proposed.

With these contributions, the thesis opens new perspectives for developing innovative computing methods. It provides new insights into modeling and solving distributed problems. Since, the systematic description of parallelism encourages and often enforces

to face the problem of critical sections – where a mutual exclusion of access is required – in early stages of solutions design.

8.3 Outlook

Further research could also be conducted to shed more light on the significance of the approach. In particular, the following topics will be studied in future research:

1. Further case studies are needed including real applications. This would allow to test the implementation of a distributed network against a formal specification by means of graph-transformational swarms rather than against informal or semi-formal models or just against the intuition of the designers.
2. The stochastic behavior of swarms should be further explored by means of probability theory, together with empirical experiments. As consequence, more general correctness results, i.e., for classes of problems and not only for specific problems, can be developed.
3. The use of tools has to be made more comfortable. At the moment, one must adapt each graph-transformational swarm separately by hand to simulate and visualize it on a graph-transformation engine or to verify properties on a SAT-solver. By fixing the syntactic features of swarm modeling, one can construct translators into the tools so that the tools run automatically on swarms and simulate and verify dynamic networks in this way.
4. It may be meaningful to translate the modeling concepts of graph-transformational swarms into explicit modeling concepts of dynamic logistic networks. In this way, modelers of networks do not need to make themselves familiar with the swarm ideas, and they could follow their intentions directly within the edifice of ideas of logistic networks.

Index

- CG , 108
- H_G , 98
- W_G , 97
- \mathcal{C} , 52
- \mathcal{X} , 53
- \mathcal{G}_Σ , 31
- \mathcal{R} , 35
- Σ , 31
- \overline{W} , 108
- $\overline{W}_{CG}^{partial}$, 107
- \overline{W}_{CG} , 107
- ant colonies, 13
- Argentine ant, 13
- binary bridge experiment, 15
- boids, 12
- complete graph, 98
- complete bipartite graph, 98
- contact condition, 33
- control condition, 52
- cooperation condition, 72
- cyle, 97
- discrete optimization problem, 98
- disjoint union, 34
- extension, 34
- foraging behavior, 13
- gluing condition, 36
- graph, 31
- graph class expression, 52
- graph morphism, 34
- graph transformation unit, 53
- graph-transformational swarm, 73
- Hamiltonian cycle, 97
- identification condition, 36
- isomorphism, 34
- loop, 31
- many eyes effect, 11
- match, 34
- parallel independence, 43
- parallel rule, 43
- pheromone, 13
- related unit, 70
- restriction, 34
- rule, 35
- school of fish, 10
- scout bees, 27
- self-organization, 10
- Short-path experiment, 16
- simple graph transformation unit, 69
- stationary members, 124
- stigmergy, 13
- stochastic control, 80
- subgraph, 33
- subtraction, 33
- swarm computation, 74
- walk, 97

Bibliography

- [1] L. Abdenebaoui and H.-J. Kreowski. Decentralized routing of automated guided vehicles by means of graph-transformational swarms. In M. Freitag, H. Kotzab, and J. Pannek, editors, *Dynamics in Logistics, Proceedings of the 5th International Conference LDIC, 2016 Bremen, Germany*, Lecture Notes in Logistics. Springer, 2016.
- [2] L. Abdenebaoui and H.-J. Kreowski. Modeling of decentralized processes in dynamic logistic networks by means of graph-transformational swarms. *Logistics Research*, 9(1):1–13, 2016.
- [3] L. Abdenebaoui, H.-J. Kreowski, and S. Kuske. Graph-transformational swarms. In S. Bensch, F. Drewes, R. Freund, and F. Otto, editors, *Fifth Workshop on Non-Classical Models for Automata and Applications - NCMA 2013, Umeå, Sweden, August 13 - August 14, Proceedings*, pages 35–50. Österreichische Computer Gesellschaft, 2013.
- [4] L. Abdenebaoui, H.-J. Kreowski, and S. Kuske. Graph-transformational swarms with stationary members. In M. L. Camarinha-Matos, A. T. Baldissera, G. Di Orio, and F. Marques, editors, *Technological Innovation for Cloud-Based Engineering Systems: 6th IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2015, Costa de Caparica, Portugal, April 13-15, 2015, Proceedings*, pages 137–144. Springer International Publishing, 2015.
- [5] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. Graph relabelling systems: a tool for encoding, proving, studying and visualizing - distributed algorithms. *Electr. Notes Theor. Comput. Sci.*, 51:93–107, 2001.
- [6] C. Blum and D. Merkle, editors. *Swarm Intelligence: Introduction and Applications*. Natural Computing Series. Springer, New York, 2008.
- [7] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.
- [8] F. M. Burnet. *The clonal selection theory of acquired immunity*. Nashville, Vanderbilt University Press, 1959.
<http://www.biodiversitylibrary.org/bibliography/8281>.

- [9] S. Camazine, N. R. Franks, J. Sneyd, E. Bonabeau, J.-L. Deneubourg, and G. Theraula. *Self-Organization in Biological Systems*. Princeton University Press, Princeton, NJ, USA, 2001.
- [10] L. N. d. Castro. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer-Verlag, London, 2002.
- [11] M. Clerc. Discrete particle swarm optimization, illustrated by the traveling salesman problem. In *New Optimization Techniques in Engineering*, volume 141 of *Studies in Fuzziness and Soft Computing*, pages 219–239. Springer Berlin Heidelberg, 2004.
- [12] E. F. Codd. *Cellular Automata*. Academic Press, New York, 1968.
- [13] M. Cook. Universality in Elementary Cellular Automata. *Complex Systems*, 15(1):1–40, 2004.
- [14] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors. *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*. Springer, 2006.
- [15] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, pages 163–245. World Scientific, Singapore, 1997.
- [16] I. D. Couzin and J. Krause. Self-Organization and Collective Behavior in Vertebrates. *Advances in the Study of Behavior*, 32:1 – 75, 2003.
- [17] L. N. de Castro and F. J. V. Zuben. Learning and optimization using the clonal selection principle. *IEEE Transactions on Evolutionary Computation, Special Issue on Artificial Immune Systems (IEEE)*, 6(3):239–251, 2002.
- [18] J. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the Argentine ant. *Journal of Insect Behavior*, 3(2):159–168, Mar. 1990.
- [19] Dijkstra, E. W. A note on two problems in connection with graphs. *Numerical Mathematics*, 1(5):269–271, 1959.
- [20] M. Dorigo, G. D. Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5:137–172, 1999.
- [21] M. Dorigo, V. Maniezzo, and A. Coloni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-part B*, 26(1):29–41, 1996.

- [22] M. Dorigo and T. Stützle. *Ant colony optimization*. MIT Press, 2004.
- [23] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, Berlin Heidelberg, 2006.
- [24] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, Berlin Heidelberg, 2006.
- [25] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, Singapore, 1999.
- [26] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, volume 7562 of *Lecture Notes in Computer Science*. Springer, 2012.
- [27] H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors. *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.
- [28] H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Graph Grammars and Their Application to Computer Science*, Berlin, 1991.
- [29] H. Ehrig, M. Pfender, and H.-J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conf. on Automata and Switching Theory*, pages 167–180, Iowa City, 1973.
- [30] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28–October 1, 2004. Proceedings*, pages 161–177, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [31] H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors. *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - - October 2, 2010. Proceedings*, volume 6372 of *Lecture Notes in Computer Science*. Springer, 2010.
- [32] K. Ehrig and H. Giese, editors. *Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 6 of *Electronic Communications of the EASST*, 2007.
- [33] A. P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. John Wiley & Sons, 2006.

- [34] M. Ermler. Towards a verification framework for haskell by combining graph transformation units and sat solving. In M. Hanus and R. Rocha, editors, *Kiel Declarative Programming Days 2013*, pages 138–152. Christian-Albrechts-Universität zu Kiel, 2013.
- [35] M. Ermler, S. Kuske, M. Luderer, and C. von Totth. A graph transformational view on reductions in np. *Electronic Communications of the EASST*, 61, 2013.
- [36] M. Gardner. The fantastic combinations of john conway’s new solitaire game "life". *Scientific American*, 223(10):120–123, Oct. 1970.
- [37] R. Geiß and M. Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd Int. Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE ’07)*, volume 5088 of *Lecture Notes in Computer Science*, pages 568–569, 2008.
- [38] S. Goss, S. Aron, J. Deneubourg, and J. Pasteels. Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989.
- [39] S. Goss, S. Aron, J. Deneubourg, and J. Pasteels. Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76(12):579–581, Dec. 1989.
- [40] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.
- [41] R. Heckel, G. Lajios, and S. Menge. Stochastic graph transformation systems. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 210–225. Springer Berlin Heidelberg, 2004.
- [42] J. Herskin and J. F. Steffensen. Energy savings in sea bass swimming in a school: measurements of tail beat frequency and oxygen consumption at different swimming speeds. *Journal of Fish Biology*, 53(2):366–376, 1998.
- [43] B. Hölldobler and E. Wilson. *The Ants*. Belknap Press of Harvard University Press, 1990.
- [44] K. Hölscher, H.-J. Kreowski, and S. Kuske. Autonomous units to model interacting sequential and parallel processes. *Fundamenta Informaticae*, 92(3):233–257, 2009.
- [45] S. Hubbard, P. Babak, S. T. Sigurdsson, and K. G. Magnússon. A model of the formation of fish schools and migrations of fish. *Ecological Modelling*, 174(4):359 – 374, 2004.
- [46] M. Hülsmann, B. Scholz-Reiter, and K. Windt. *Autonomous Cooperation and Control in Logistics*. Springer, Berlin Heidelberg, 2011.
- [47] J. Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334:3–33, Apr. 2005.

- [48] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks (ICNN'95)*, volume 4, pages 1942–1948, Perth, Western Australia, Nov. 1995. IEEE.
- [49] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4. IEEE, Nov. 1995.
- [50] J. Kennedy and R. C. Eberhart. *Swarm Intelligence*. Evolutionary Computation Series. Morgan Kaufman, San Francisco, 2001.
- [51] O. Kniemeyer. *Design and implementation of a graph grammar based language for functional-structural plant modelling*. PhD thesis, Brandenburg University of Technology, 2008.
- [52] H.-J. Kreowski. *Manipulationen von Graphmanipulationen*. PhD thesis, Technische Universität Berlin, 1977.
- [53] H.-J. Kreowski, R. Klempien-Hinrichs, and S. Kuske. Some essentials of graph transformation. In Z. Ésik, C. Martín-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 229–254. Springer, 2006.
- [54] H.-J. Kreowski, R. Klempien-Hinrichs, and S. Kuske. Some essentials of graph transformation. In Z. Esik, C. Martin-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 229–254. Springer, Berlin Heidelberg, 2006.
- [55] H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units — a step into GRACE. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 89–108, 1996.
- [56] H.-J. Kreowski and S. Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.
- [57] H.-J. Kreowski and S. Kuske. Autonomous units and their semantics - the concurrent case. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 102–120. Springer, 2010.
- [58] H.-J. Kreowski and S. Kuske. Graph multiset transformation: a new framework for massively parallel computation inspired by dna computing. *Natural Computing*, 10(2):961–986, 2011.
- [59] H.-J. Kreowski, S. Kuske, and G. Rozenberg. Graph transformation units – an overview. In P. Degano, R. D. Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 57–75. Springer, 2008.

- [60] H. Kunz and C. K. Hemelrijk. Artificial fish schools: Collective effects of school size, body size, and body form. *Artificial Life*, 9(3):237–253, 2003.
- [61] S. Kuske. More about control conditions for transformation units. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *lncs*, pages 323–337, 2000.
- [62] S. Kuske. *Transformation Units—A structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.
- [63] S. Kuske and M. Luderer. Autonomous units for solving the capacitated vehicle routing problem based on ant colony optimization. *ECEASST*, 26, 2010.
- [64] S. Kuske, M. Luderer, and H. Tönnies. Autonomous units for solving the traveling salesperson problem based on ant colony optimization. In H.-J. Kreowski, B. Scholz-Reiter, and K.-D. Thoben, editors, *Dynamics in Logistics*, pages 289–298. Springer, Berlin Heidelberg, 2011.
- [65] T. Le-Anh and M. D. Koster. A review of design and control of automated guided vehicle systems. *European Journal of Operational Research*, 171(1):1 – 23, 2006.
- [66] H. R. Loureno and D. Serra. Adaptive search heuristics for the generalized assignment problem. *Mathware and Soft Computing*, 9(2-3):209–234, Dec. 10 2002.
- [67] M. Luderer. *Control Conditions for Transformation Units : Parallelism, As-long-as-possible, and Stepwise Control*. PhD thesis, Universität Bremen: Informatik/-Mathematik, 2016.
- [68] Y. Marinakis and M. Marinaki. A hybrid multi-swarm particle swarm optimization algorithm for the probabilistic traveling salesman problem. *Computers & OR*, 37(3):432–442, 2010.
- [69] Y. Métivier and E. Sopena. Graph relabelling systems : a general overview. *Computers and artificial intelligence*, 16(2):167–185, 1997.
- [70] A. Moraglio and J. Togelius. Geometric particle swarm optimization for the sudoku puzzle. In H. Lipson, editor, *GECCO*, pages 118–125. ACM, 2007.
- [71] S. Olariu and A. Y. Zomaya. *Handbook of Bioinspired Algorithms and Applications*. Chapman & Hall/CRC, 2005.
- [72] R. S. Olson, P. B. Haley, F. C. Dyer, and C. Adami. Exploring the evolution of a trade-off between vigilance and foraging in group-living organisms. *Royal Society Open Science*, 2(9), 2015.
- [73] B. Partridge and T. Pitcher. The sensory basis of fish schools: Relative roles of lateral line and vision. *Journal of Comparative Physiology*, 135(4):315–325, 1980.
- [74] B. L. Partridge. The structure and function of fish schools. *Scientific American*, pages 114–123, June 1982.

- [75] M. Pedemonte, S. Nesmachnow, and H. Cancela. A survey on parallel ant colony optimization. *Applied Soft Computing*, 11(8):5181 – 5197, 2011.
- [76] H.-O. Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals: New Frontiers of Science*. Springer, 1992.
- [77] A. Perna, B. Granovski, S. Garnier, S. C. Nicolis, M. Labédan, G. Theraulaz, V. Fourcassié, and D. J. T. Sumpter. Individual Rules for Trail Pattern Formation in Argentine Ants (*Linepithema humile*). *PLoS Computational Biology*, 8(7):e1002592, 07 2012.
- [78] J. L. Pfaltz and A. Rosenfeld. Web grammars. In *Proc. Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.
- [79] D. T. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, and M. Zaidi. The Bees Algorithm, A Novel Tool for Complex Optimisation Problems. In *Proceedings of the 2nd International Virtual Conference on Intelligent Production Machines and Systems (IPROMS 2006)*, pages 454–459. Elsevier, 2006.
- [80] T. Pitcher. Functions of shoaling behaviour in teleosts. In T. Pitcher, editor, *The Behaviour of Teleost Fishes*, pages 294–337. Springer US, 1986.
- [81] T. Pitcher and B. Partridge. Fish school density and volume. *Marine Biology*, 54(4):383–394, 1979.
- [82] R. Poli. Analysis of the publications on the applications of particle swarm optimisation. *Journal of Artificial Evolution and Applications*, pages 1–10, 2008.
- [83] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization – an overview. *Swarm Intelligence*, 1(1):33–57, 2007.
- [84] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Syst. Sci.*, 5(6):560–595, 1971.
- [85] D. V. D. i. V. Radakov. *Schooling in the ecology of fish*. New York : J. Wiley, 1973. "A Halsted Press book."
- [86] P. Rendell. A fully universal turing machine in conway’s game of life. *Journal of Cellular Automata*, 8(1-2):19–38, 2013.
- [87] C. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4), July 1987.
- [88] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, Singapore, 1997.
- [89] G. Rozenberg, editor. *Handbook on Graph Grammars and Computing by Graph Transformation Vol. 1 Foundations*. World Scientific, Singapore, 1997.

- [90] G. Rozenberg, H. Ehrig, et al., editors. *Handbook on Graph Grammars and Computing by Graph Transformation 3 (Concurrency)*. World Scientific, Singapore, 1999.
- [91] H. J. Schneider. Formal systems for structure manipulation. In W. Händler, J. Weizenbaum, and D. Bitzer, editors, *Display Use for Man-Machine Dialog*, München, 1971. Hanser.
- [92] C. Schwarz and J. Sauer. Towards decentralised agv control with negotiations. In K. Kersting and M. Toussaint, editors, *Proceedings of the Sixth Starting AI Researchers Symposium*, volume 241 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2012.
- [93] N. Smolic-Rocak, S. Bogdan, Z. Kovacic, and T. Petrovic. Time windows based dynamic routing in multi-AGV systems. *IEEE T. Automation Science and Engineering*, 7(1):151–155, 2010.
- [94] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
- [95] J. C. Svendsen, J. Skov, M. Bildsoe, and J. F. Steffensen. Intra-school positional preference and reduced tail beat frequency in trailing positions in schooling roach under experimental conditions. *Journal of Fish Biology*, 62(4):834–846, 2003.
- [96] G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance: Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, pages 446–453, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [97] F. Taghaboni-dutta and J. M. A. Tanchoco. Comparison of dynamic routeing techniques for automated guided vehicle system. *International Journal of Production Research*, 33(10):2653–2669, 1995.
- [98] A. ter Mors, C. Witteveen, J. Zutt, and F. A. Kuipers. Context-aware route planning. In J. Dix and C. Witteveen, editors, *Multiagent System Technologies, 8th German Conference, MATES 2010, Leipzig, Germany*, volume 6251 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 2010.
- [99] J. F. Traniello. foraging strategies of ants. *Annual Review of Entomology*, 34:191–210, 1989.
- [100] I. F. Vis. Survey of research in the design and control of automated guided vehicle systems. *European Journal of Operational Research*, 170(3):677 – 709, 2006.
- [101] J. von Neumann. *The general and logical theory of automata*, pages 1–41. Wiley, Pasadena CA, 1951.

- [102] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966. Edited and completed by Arthur W. Burks.
- [103] D. Weyns, T. Holvoet, K. Schelfhout, and J. Wielemans. Decentralized control of automatic guided vehicles: Applying multi-agent systems in practice. In *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, pages 663–674, New York, NY, USA, 2008. ACM.
- [104] S. Wolfram. *A New Kind of Science*. Wolfram Media Inc., 2002.
- [105] Y. Zhang, S. Wang, and G. Ji. A comprehensive survey on particle swarm optimization algorithm and its applications. *Mathematical Problems in Engineering*, 2015:38, 2015.