entropy

Article

A Graph-Transformational Approach to Swarm Computation

Larbi Abdenebaoui [1], Hans-Jörg Kreowski [2,*] and Sabine Kuske [2]

[1] OFFIS—Institute for Information Technology, Escherweg 2, 26122 Oldenburg, Germany; larbi.abdenebaoui@offis.de
[2] Department of Computer Science, University of Bremen, P.O. Box 330440, D-28334 Bremen, Germany; kuske@uni-bremen.de
* Correspondence: kreo@informatik.uni-bremen.de

Abstract: In this paper, we propose a graph-transformational approach to swarm computation that is flexible enough to cover various existing notions of swarms and swarm computation, and it provides a mathematical basis for the analysis of swarms with respect to their correct behavior and efficiency. A graph transformational swarm consists of members of some kinds. They are modeled by graph transformation units providing rules and control conditions to specify the capability of members and kinds. The swarm members act on an environment—represented by a graph—by applying their rules in parallel. Moreover, a swarm has a cooperation condition to coordinate the simultaneous actions of the swarm members and two graph class expressions to specify the initial environments on one hand and to fix the goal on the other hand. Semantically, a swarm runs from an initial environment to one that fulfills the goal by a sequence of simultaneous actions of all its members. As main results, we show that cellular automata and particle swarms can be simulated by graph-transformational swarms. Moreover, we give an illustrative example of a simple ant colony the ants of which forage for food choosing their tracks randomly based on pheromone trails.

Keywords: swarm computation; graph transformation; cellular automata; particle swarms

Citation: Abdenebaoui, L.; Kreowski, H.-J.; Kuske, S. A Graph-Transformational Approach to Swarm Computation. *Entropy* **2021**, *23*, 453. https://doi.org/10.3390/e23040453

Academic Editor: Leonardo Vanneschi

Received: 18 February 2021
Accepted: 31 March 2021
Published: 12 April 2021

1. Introduction

The idea of swarm computation is to design systems that mimic the problem-solving behavior of swarms in nature like ant colonies, bee hives, bird flocks, fish schools, etc. One encounters quite a variety of swarm concepts and swarm algorithms in the literature (see, e.g., [1–9]). Moreover, there are several general computational approaches like cellular automata, particle swarms and ant colony optimization that are subsumed under the heading of swarm intelligence. In this paper, we propose graph-transformational swarms as a unifying framework using the methods of graph transformation. The notion of graph-transformational swarms is flexible enough to cover a variety of swarm concepts and provides a mathematical basis for the analysis of swarms with respect to correctness and efficiency. The hope is that different models of swarm computation can be better compared with each other within a common framework and that results for one model can be carried over to other models more easily. Moreover, the graph-transformational approach allows to employ graph-transformation tools for simulation, model checking and SAT solving in a standardized way.

A graph-transformational swarm consists of an arbitrary number of members of a finite number of different kinds. The members act simultaneously in a common environment which is represented as a graph. Moreover, there may be a cooperation condition to regulate the interaction and cooperation of the members as well as a goal to be reached. Kinds and members are modeled as graph transformation units (see, e.g., [10]) 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. First ideas of graph-transformational swarms are presented by [11] and [12] where typical applications of ant colony optimization algorithms are modeled, but a general definition of graph transformational swarms is missing. A short draft version of this paper appeared as [13]. A good part of the paper is also integrated into the first author's PhD thesis [14].

The paper is organized in the following way. In Section 2, the basic notions of graph transformation are recalled. Section 3 introduces graph-transformational swarms. In Section 4, an illustrating example is given: a simple ant colony the ants of which forage for food in a pheromone-driven manner. To demonstrate the power of our approach, we embed cellular automata in Section 5 and particle swarms in Section 6.

2. Graph Transformation

In this section, we recall the basic elements of graph transformation as far as needed in this paper (for more details, see, e.g., [15–17]). We consider directed edge-labeled graphs and their derivation by applications 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 an additional independence condition. Moreover, we use the notion of graph transformation units which comprise a set of rules and a control condition. Such a unit is a computational device that models the derivation of graphs while the control condition is obeyed. Units are used as members of swarms, and the parallelism makes sure that the members can act simultaneously (cf. Section 3).

2.1. Directed Edge-Labeled Graphs

Let Σ be a set of labels with $* \in \Sigma$. 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*, $s, t \colon E \to V$ and $l \colon E \to \Sigma$ are mappings assigning a *source* $s(e)$, a *target* $t(e)$ and a *label* $l(e)$ to every edge $e \in E$.

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* resp. An edge with label $*$ represents an *unlabeled edge*. In drawings of graphs, the label $*$ is omitted. 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 over Σ is denoted by $\mathcal{G}_\Sigma$.

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

For graphs $G, H \in \mathcal{G}_\Sigma$, a *graph morphism* $g \colon G \to H$ is a pair of mappings $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ 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 the mappings g_V and g_E are inclusions, then G is called a *subgraph* of H, denoted by $G \subseteq H$. The *match* of G with respect to the morphism g is the subgraph $g(G) \subseteq H$.

2.2. Graph Transformation Rules

A *rule* $r = (L, K, R)$ consists of three graphs $L, K, R \in \mathcal{G}_\Sigma$ such that $L \supseteq K \subseteq R$. A *rule with positive context* $r = (C, L, K, R)$ consists of four graphs C, L, K, and R such that (L, K, R) is a rule and $L \subseteq C$. If C equals L, it is omitted in r. The components C, L, K, and R are called *positive context*, *left-hand side*, *gluing graph*, and *right-hand side*, respectively. Sample rules are always presented with the inclusion symbols so that left-hand side, gluing graph, right-hand side, and a possible positive context are clear from their positions. In order to avoid too much technical detail, we assume that the node sets of L and K are equal. This means that rule applications do not delete nodes. Figure 1 shows the rule *found* a variant of which is used in Section 4 for modeling a simple ant colony.

The gluing graph consists of two nodes, say u and v, as well as an unlabeled edge from u to v and a *food*-loop at v. The left-hand side consists of the gluing graph and an A-edge from u to v. The right-hand side consists of the gluing graph and an A^+-edge from v to u as well as an ϵ-edge from u to v.

Intuitively, the application of a rule (L, K, R) replaces an occurrence of L in some graph by R such that the occurrence of K is kept. Hence, the application of the rule *found* reverses an A-edge into an A^+-edge provided that it is attached to a node with a *food*-loop. Additionally, it inserts an ϵ-edge.

A rule with positive context (C, L, K, R) is applied in the same way as (L, K, R) provided that the occurence of L is located within an occurrence of C. If the left-hand-side of the rule *found* is regarded as positive context, we can remove the *food*-loop as well as the unlabeled edge from the remaining three rule components, because they are not changed. The result is displayed in Figure 2 where the two nodes of the gluing graph are numbered to fix their inclusion into the other graphs. It is worth noting that the rule in Figure 1 and the rule in Figure 2 are semantically equivalent.

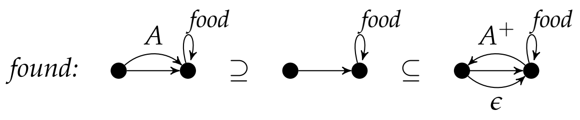


Figure 1. A graph transformation rule.

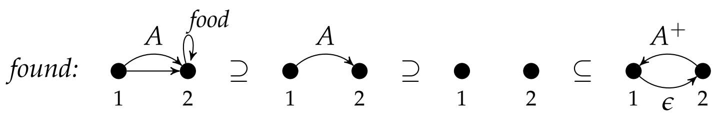


Figure 2. A graph transformation rule with positive context.

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

1. Choose a match $g(L)$ of L in G subject to the *identification condition*, which requires that those items that are identified via g belong to the gluing graph K, i.e., $g_E(e) = g_E(e')$ for $e, e' \in E_L$ implies $e = e'$ or $e, e' \in E_K$. (Without the identification condition, the Parallelization Theorem below would not hold.)
2. Remove the edges of $g_E(E_L) - g_E(E_K)$ and call the resulting graph Z.
3. Add the right-hand side R to Z by gluing Z with R in $g(K)$ yielding the graph H with $V_H = V_Z \uplus (V_R - V_K)$ and $E_H = E_Z \uplus (E_R - E_K)$. The edges of Z keep their labels, sources, and targets so that $Z \subseteq H$. The edges of R keep their labels; they also keep their sources and targets provided that those belong to $V_R - V_K$. Otherwise, they are redirected to the image of their original source or target, i.e., $s_H(e) = g(s_R(e))$ for $e \in E_R - E_K$ with $s_R(e) \in V_K$, and $t_H(e) = g(t_R(e))$ for $e \in E_R - E_K$ with $t_R(e) \in V_K$.

A rule with positive context $r = (C, L, K, R)$ is applied to G in the same way provided that the morphism $g\colon L \to G$ can be extended to C. Figure 3 shows two applications of *found*.

An application of r to G w.r.t. the graph morphism g is denoted by $G \underset{r}{\Longrightarrow} H$. It is called a *direct derivation* from G to H. The subscript r may be omitted if it is clear from the context. The sequential composition of direct derivations $G = G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n = H$ ($n \in \mathbb{N}$) is called a *derivation* from G to H. As usual, the derivation from G to H can also be denoted by $G \overset{n}{\underset{P}{\Longrightarrow}} H$ where $\{r_1, \ldots, r_n\} \subseteq P$, or just by $G \overset{*}{\underset{P}{\Longrightarrow}} H$. The string $\langle r_1, \ldots, r_n \rangle$ is the *application sequence* of the derivation. Figure 4 shows a derivation with application sequence $\langle found, found \rangle$.

Instead of applying $found$ to the left upper A-edge and then to the right upper one, one can interchange the order which yields the same result with a different intermediate graph.

In the following, the class of all rules (with and without positive context) is denoted by $\mathcal{R}$.

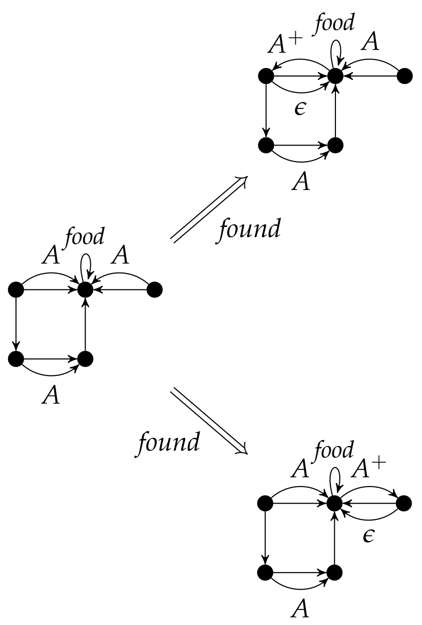


Figure 3. Two rule applications.

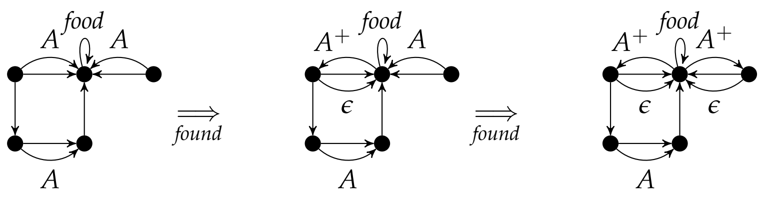


Figure 4. A derivation.

2.3. Parallel Rule Application

Let $r_i = (C_i, L_i, K_i, R_i) \in \mathcal{R}$ for $i = 1, \ldots, n$. Then the *parallel rule* $p = \sum_{i=1}^{n} r_i = (\sum_{i=1}^{n} C_i, \sum_{i=1}^{n} L_i, \sum_{i=1}^{n} K_i, \sum_{i=1}^{n} R_i)$ is given by the disjoint unions of the components. Figure 5 shows the parallel rule *found* + *found*. It can be applied to the left graph of Figure 4, if the A-edges are not identified (otherwise, the identification condition would be violated). The result is equal to the right graph of Figure 4 (see also Figure 6).

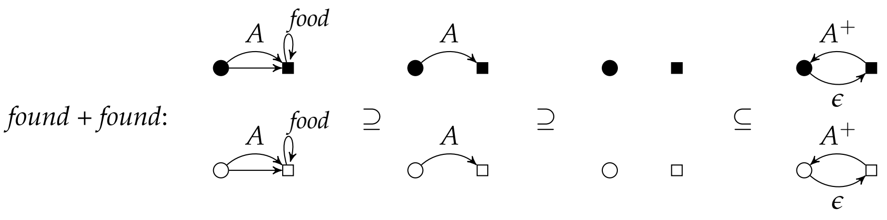


Figure 5. A parallel rule.

Let $r = (C, L, K, R)$ and $r' = (C', L', K', R')$ be two rules and let $G \underset{r}{\Longrightarrow} H$ and $G \underset{r'}{\Longrightarrow} H'$ be two direct derivations w.r.t. the morphisms $g\colon L \to G$ and $g'\colon L' \to G$. Then the direct derivations are *parallel independent* if the corresponding matches intersect in gluing items only, i.e., $g_V(V_L) \cap g'_V(V_{L'}) \subseteq g_V(V_K) \cap g'_V(V_{K'})$ and $g_E(E_L) \cap g'_E(E_{L'}) \subseteq g_E(E_K) \cap g'_E(E_{K'})$.

The application of parallel rules and parallel independence are closely related as is shown by the well-known Paralllelization Theorem (see, e.g., [16,17] and Chapter 2 of [15]). This result is the basis of the simultaneous actions of members of graph-transformational swarms as introduced in the next section.

Fact 1 (Parallelization Theorem)**.** For $i = 1, \dots, n$, let $r_i = (C_i, L_i, K_i, R_i) \in \mathcal{R}$ and let $p = (C, L, K, R) = \sum_{i=1}^{n} r_i$ be the corresponding parallel rule. Then the following hold.

1. Let $G \underset{p}{\Longrightarrow} X$ be a direct derivation w.r.t. $g\colon L \to G$. Then there are direct derivations $G \underset{r_i}{\Longrightarrow} H_i$ with the matching morphisms $g_i = g|L_i$ that are pairwise parallel independent where the morphism $g|L'\colon L' \to G$ denotes the restriction of g to L'. for $g\colon L \to G$ and $L' \subseteq L$.
2. Let $G \underset{r_i}{\Longrightarrow} H_i$ for $i = 1, \dots, n$ be direct derivations w.r.t. $g_i\colon L_i \to G$. Let each two of them be parallel independent. Then there is a direct derivation $G \underset{p}{\Longrightarrow} X$ w.r.t. $g\colon L \to G$ defined by $g|L_i = g_i$ for $i = 1, \dots, n$.

The theorem still holds for an infinite family $r_i \in \mathcal{R}$ with $i \in \mathbb{N}$.

According to the Parallelization Theorem, the rule components of *found+found* in Figure 6 can be applied separately to the left graph and are parallel independent. Conversely, these two parallel independent applications of *found* can be executed in parallel.

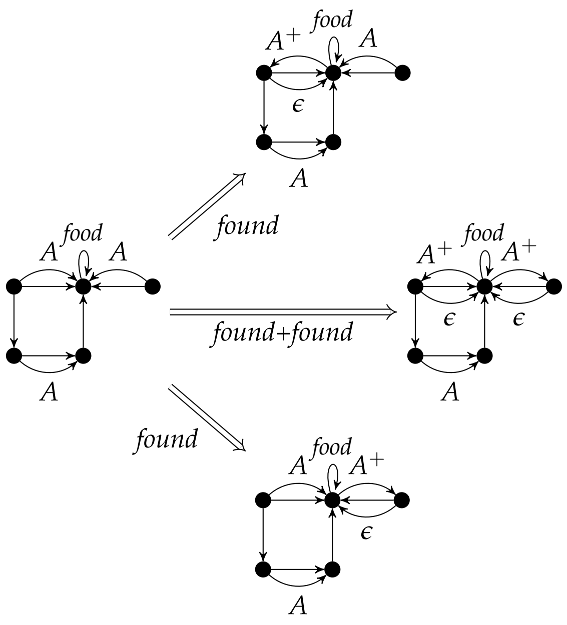


Figure 6. Two parallel independent rule applications and their parallelization.

2.4. Control Conditions and Graph Class Expressions

Control conditions can reduce the nondeterminism of rule application. In more detail, each control condition C is defined over a finite set P of rules and specifies a set $SEM(C)$ of derivations. 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 $C_1, C_2 \in \mathcal{C}$.

A typical control condition is a priority relation $>$ on a set P of rules meaning that a rule $r \in P$ can only be applied if no other rule with higher priority is applicable. Another often used control condition is a regular expression over P. By definition, the constants *empty*, *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. 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; $e_1|e_2$ allows a derivation if e_1 or e_2 allows it; e^* allows a derivation if it is a sequence of sub-derivations each allowed by e. The expression $r \in P$ requires that r is applied; *lambda* allows any derivation of length 0; *empty* forbids any derivation. Alternatively to r^*, $r!$ is used. It requests that r is applied as long as possible and not arbitrarily often.

All these examples of control conditions and their satisfaction apply not only to derivations over P but also to derivations of the form

$$G_0 \underset{r_1+r'_1}{\Longrightarrow} G_1 \underset{r_2+r'_2}{\Longrightarrow} \cdots \underset{r_n+r'_n}{\Longrightarrow} G_n$$

with $r_1, \dots . r_n \in P$ and $r'_1, \dots, r'_n \in P'$ if $\langle r_1, \dots, r_n \rangle$ is an application sequence in the case of regular expressions or if, in the case of priorities for all $i = 1, \dots, n$, $G_{i-1} \underset{r_i+r'_i}{\Longrightarrow} G_i$ implies $r_i \geq \hat{r}$ for all $\hat{r} \in P$ applicable to G_{i-1}.

Graph class expressions restrict the class $\mathcal{G}_\Sigma$ to subclasses, i.e., each graph class expression X specifies a set $SEM(X) \subseteq \mathcal{G}_\Sigma$. The class of all graph class expressions is denoted by $\mathcal{X}$. Typical examples of graph class expressions are graph properties like *unlabeled* with $SEM(unlabeled) = \mathcal{G}_{\{*\}}$ or *simple* with $SEM(simple) = \{(V, E, pr_1, pr_2, pr_3) \mid E \subseteq V \times V \times \Sigma\}$ where pr_i is the projection to the i-th component for $i = 1, 2, 3$. Moreover, each graph $G \in \mathcal{G}_\Sigma$ is a graph class expression with $SEM(G) = \{G\}$. We also use $required(X)$ and $op(X)$ for $X \in \mathcal{X}$ as graph class expressions. $SEM(required(X))$ contains all graphs with a subgraph in $SEM(X)$. $SEM(op(X))$ for some graph operator *op* contains all graphs obtained by the application of the operator to graphs in $SEM(X)$. Explicit examples of such operators are *nest-looping* and $food^*$*-looping*. Applied to $G \in \mathcal{G}_\Sigma$, the first operator adds one *nest*-loop to some node, and the second operator adds an arbitrary number of $food$-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 $X_1, X_2 \in \mathcal{X}$. Further graph class expressions are introduced where needed.

2.5. Graph Transformation Units

In the following we introduce a special case of graph transformation units, which is suitable for our purposes.

A *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. The *semantics* of gtu consists of all derivations $G \overset{*}{\underset{P}{\Longrightarrow}} H$ allowed by C.

A unit gtu is *related* to a unit gtu_0 if gtu is obtained from gtu_0 by relabeling. For a mapping $rel\colon \Sigma \to \Sigma$, 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 $RU(gtu_0)$.

Each set $P \subseteq \mathcal{R}$ of rules induces a graph transformation unit specified by $gtu(P) = (P, free)$ where *free* allows all derivations. For $gtu(\{p\})$ with $p \in \mathcal{R}$ we write $gtu(p)$ for short.

3. Graph-Transformational Swarms

In this section, we introduce 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, we also allow multidimensional swarms 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 $(n_1, \dots, n_l) \in \mathbb{N}^l_{>0}$, 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, where $\mathbb{N}_{>0} = \mathbb{N} - \{0\}$ and $[n] = \{1, \dots, n\}$. While a kind is specified as a 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 requiring 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 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, we allow to provide a swarm with an additional cooperation condition. Finally, a swarm may have a goal given by a graph class expression like the initial graphs are specified by such an expression. A computation is considered to be successful if an environment is reached that meets the goal.

Definition 1 (swarm). *A swarm is a system $S = (I, K, s, m, c, g)$ where I is a graph class expression specifying the set of initial environments, K is a finite set of graph transformation units, called kinds, s associates a size vector $s(k) \in \mathbb{N}_{>0}^{d(k)}$ 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 $(m(k)_i)_{i \in [s(k)]}$ with each kind $k \in K$ with $m(k)_i \in RU(k)$ for all $i \in [s(k)]$, c is a control condition called cooperation condition, and g is a graph class expression specifying the goal. For $s = (n_1, \ldots, n_l) \in \mathbb{N}_{>0}^{l}$ and some $l \geq 1$, $[s] = \{(i_1, \ldots, i_l) \mid i_j \in [n_j], j \in [l]\}$.*

A swarm may be represented schematically as in Figure 7 where $s_i = s(k_i)$ and $m_i = m(k_i)$ for $i \in [n]$.

name

`initial:`	I
`kinds :`	$k_1, \ldots, k_n$
`size :`	$s_1, \ldots, s_n$
`members:`	$m_1, \ldots, m_n$
`coop:`	c
`goal:`	g

Figure 7. The schematic representation of a swarm.

Definition 2 (swarm computation). *A swarm computation is a derivation*

$$G_0 \underset{p_1}{\Longrightarrow} G_1 \underset{p_2}{\Longrightarrow} \cdots \underset{p_q}{\Longrightarrow} G_q$$

such that $G_0 \in SEM(I)$, $p_j = \sum_{k \in K} \sum_{i \in [s(k)]} r_{jki}$ with a rule r_{jki} of $m(k)_i$ for each $j \in [q]$, $k \in K$ and $i \in [s(k)]$, and c and the control conditions of all members are satisfied. For the satisfaction of the control condition of a unit, confer the definition for parallel derivations in Section 2.4.

That all members must provide a rule to a computational step, is a strong requirement 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 stops the whole swarm. To avoid this global effect of a local situation, we assume that each member has the empty rule $(\emptyset, \emptyset, \emptyset)$ in addition to its other rules. The empty rule gets 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.

To enhance the feasibility of the swarm concept, we allow also unbounded sizes, denoted by $\mathbb{N}$ or $\mathbb{Z}$. In this case, we allow only computations 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 Section 5.

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. The graph-transformational approach offers some 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 Parallelization Theorem establishes a 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., [18]). In contrast to that, the simultaneous actions of members of graph-transformational swarms is 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.

In the next three sections, we make an attempt to demonstrate the stated advantages by modeling three typical approaches to swarm computation.

4. A Simple Ant Colony

In this section, we illustrate the notion of graph-transformational swarms by modeling an ant colony the ants of which forage for food by mean of a simple pheromone mechanism. The sample graph-transformational swarm is presented in Figure 8.

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. We assume *nest*-$food$-connectedness meaning that the paths from the *nest*-looped node to some $food$-looped node visit all nodes. Moreover, we assume that the underlying environment graph is simple meaning that there are no parallel pheromone-labeled edges. This class of graphs is denoted by $(nest \,\&\, food^*)\text{-}looping(simple \,\&\, pheromone\text{-}labeled \,\&\, nest\text{-}food\text{-}connected)$. During swarm computations further edges appear and disappear.

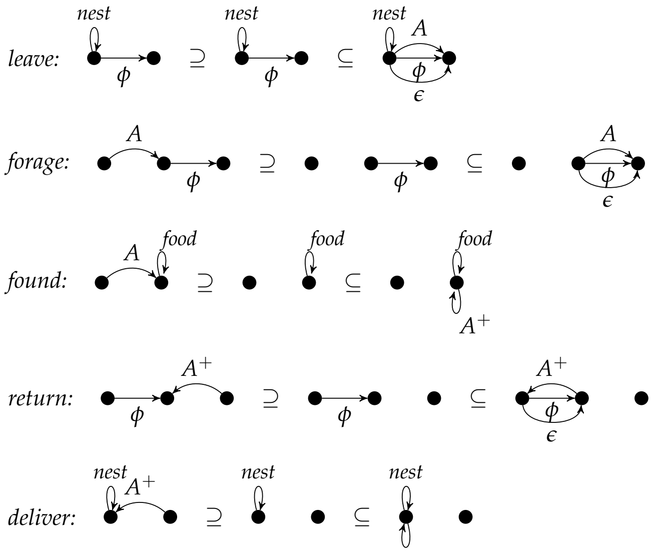
The kind *ant* defines the potential activities of an ant by means of five rules and some priorities. It can *leave* the nest by placing an A-edge and an ϵ-labeled edge in parallel to a pheromone-labeled edge with the *nest*-looped node as source. Then it can *forage* for food by walking through the 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 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. 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*.

simple ant colony

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

ant

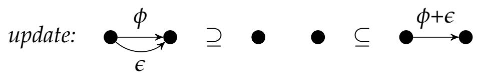
`rules:`



`control:` $leave < forage \mid found \mid return \mid deliver$
$forage < found$
$return < deliver$

update

`rules:`



`control:` $\| \; update \; \|!$

Figure 8. The swarm *simple ant colony* with the kinds *ant* and *update*.

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 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 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 $i = 1, \ldots, n$ where n is the chosen size of the ant colony. All other labels are kept. 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. But there is one restriction given by the cooperation condition. It requires that ants act *pheromone-driven* meaning 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 $e_1, \ldots, e_k$ with pheromone values $\phi_1, \ldots, \phi_k$ 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.

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 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 how 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? We assume 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 assuming in addition 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 before 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, we have kept it 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.

We have implemented the *simple ant colony* swarm in the graph transformation tool GrGen.NET [19]. An experimental computation with a swarm of 20 ants is documented in Figure 9. For a better visualization, we omit the labels of the ants and replace the loops representing the food stock 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_2 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 in two groups from 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 from 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 results 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 in three subgroups when arriving in 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 starts slowly to be preferred. In H_{28} the ants have performed 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.

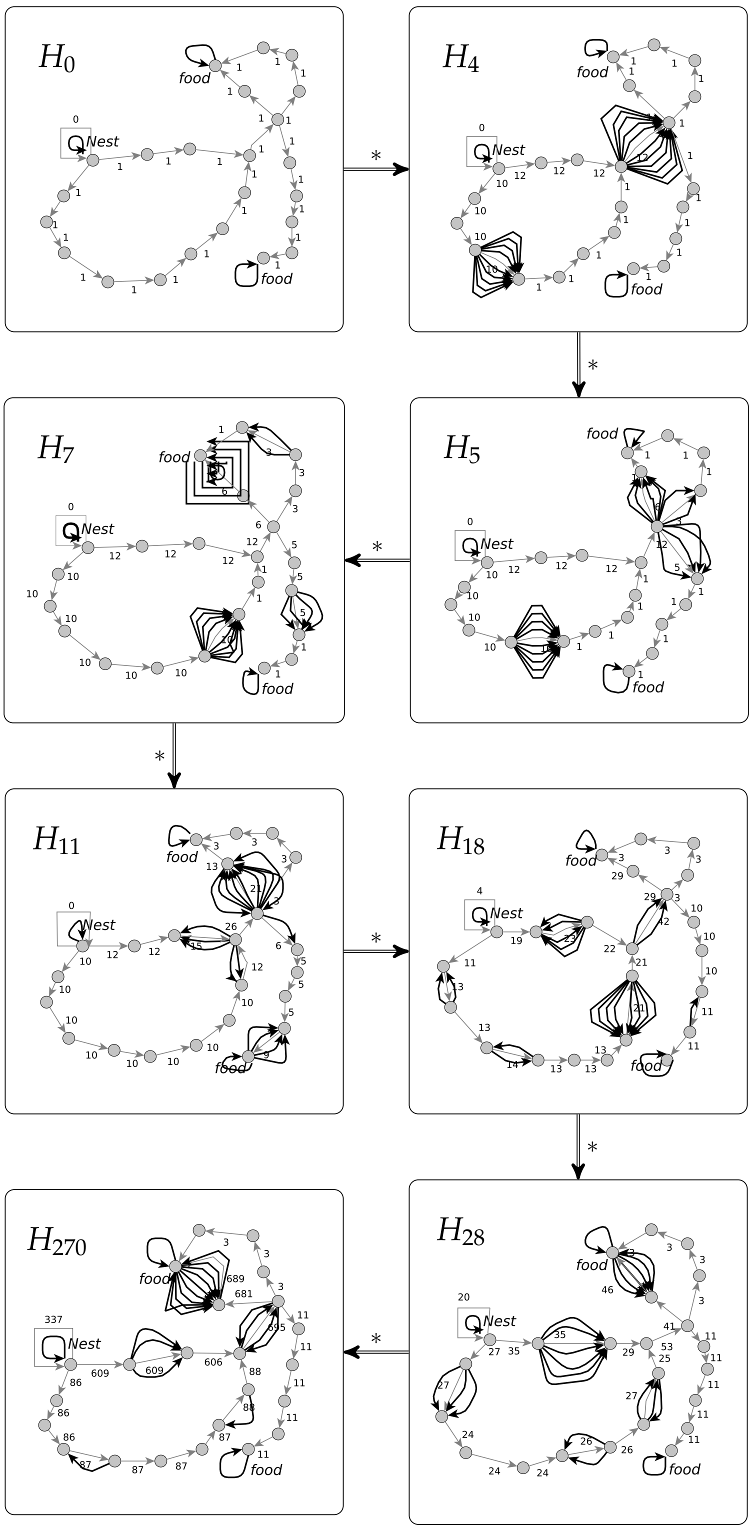


Figure 9. A sample computation of the *simple ant colony* swarm.

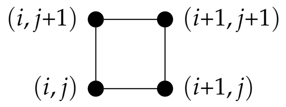
Our ant colony model is meant to exemplify how the features of graph-transformational swarms look like and work. How such models can be turned into applications that solve concrete optimization problems can be seen in [11,12].

5. Cellular Automata

Cellular automata are computational devices with massive parallelism known for many decades see, e.g., [20–24]. They are also considered as typical representatives of swarm computation [2]. 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 $N = (N_1, \ldots, N_k) \in (\mathbb{Z} \times \mathbb{Z})^k$ where the neighbor cells of (i,j) are given by the translations $(i,j) + N_1, \ldots, (i,j) + N_k$. 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 $d\colon COLOR \times COLOR^k \to COLOR$. 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 $S\colon \mathbb{Z} \times \mathbb{Z} \to COLOR$ and the follow-up configuration S' of S is defined by

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

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 1. *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., $d\colon COLOR \times COLOR^2 \to COLOR$ *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 10 after 50 steps.*

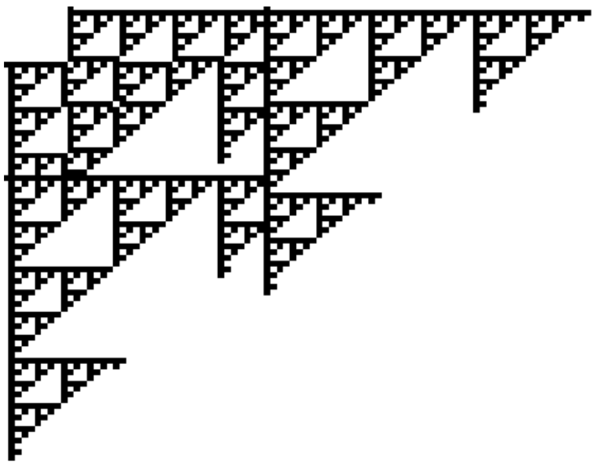


Figure 10. A pictorial representation of the configuration S_{50}.

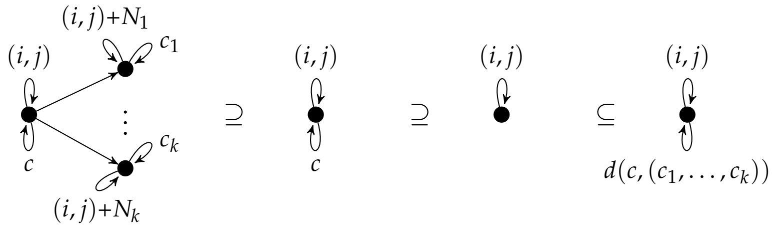
Starting with a single black cell, SIER iterates the Sierpinski gadget (cf., e.g., [25]).

Cellular automata can be considered as graph-transformational swarms. Let CA be a cellular automaton with the neighborhood vector

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

the set of colors $COLOR$ and the transition function $d\colon COLOR \times COLOR^k \to COLOR$. Then a configuration $S\colon \mathbb{Z} \times \mathbb{Z} \to COLOR$ 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., $d(S((i,j)),(S((i,j)+N_1),\ldots,S((i,j)+N_k))) \neq S(i,j)$, 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 $p = 1,\ldots,k$. 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 $d(c,(c_1,\ldots,c_k))$-loop at the node with the (i,j)-loop if and only if, for $l = 1,\ldots,k$, the neighbor with the $(i,j)+N_l$-loop has also a c_l-loop. All other c-loops are kept. This means that $H = gr(N,S')$. Summarizing, each cellular automaton can be transformed into a graph-transformational swarm such that the following correctness result holds.

Theorem 1. *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 $gr(N,S)) \Longrightarrow gr(N,S')$ 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.

6. Particle Swarm Optimization

Particle swarm optimization is one of the major approaches to swarm intelligence one encounters in the literature in various variants (see, e.g., [26–30]) In this section, we model 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 $f\colon \mathbb{R}^d \to \mathbb{R}$ and a neighborhood $N\colon \mathbb{R}^d \to \mathcal{P}(\mathbb{R}^d)$ (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 $U_t(0,\phi_1)$ 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., [28,29].

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 $f\colon \mathbb{Z}^d \to \mathbb{Z}$, the neighborhood $N\colon \mathbb{Z}^d \to \mathcal{P}(\mathbb{Z}^d)$, 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 Figure 11.

swarm(PS)

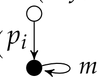
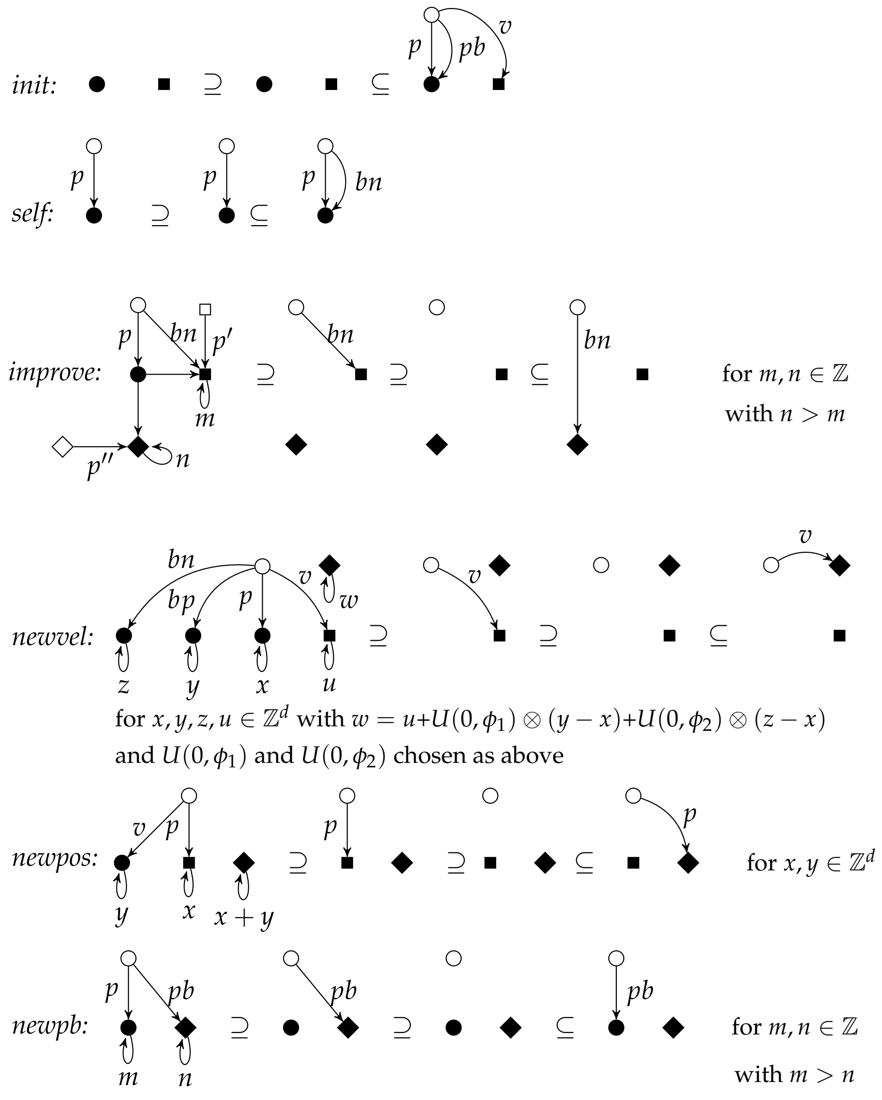
`initial:`	$space(N, f)$
`kinds :`	*particle*
`size :`	n
`members:`	$particle_i$ for $i \in [n]$
`coop:`	$synchronize(self, newvel)$
`goal:`	$required($  $\mid i \in [n], m \geq b)$

Figure 11. 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 12. (For technical simplicity, we assume $d > 1$).

particle
`rules:`



for $x, y, z, u \in \mathbb{Z}^d$ with $w = u + U(0, \phi_1) \otimes (y - x) + U(0, \phi_2) \otimes (z - x)$
and $U(0, \phi_1)$ and $U(0, \phi_2)$ chosen as above

`control:` $init\ ;\ (self\ ;\ improve!\ ;\ newvel\ ;\ newpos\ ;\ try(newpb))^*$

Figure 12. The unit *particle*.

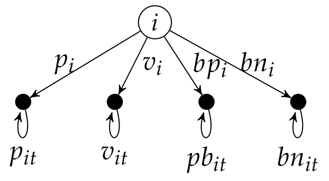
The member $particle_i$ for $i \in [n]$ is obtained by indexing p, v, pb and bn with i. All other labels are kept variable with $p', p'' \in \{p_1, \cdots, p_n\}$ in particular. Due to the control condition, the rule *init* is applied first and then never again. It chooses two points x and y, generates 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 are 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 reach 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 there is a one-to-one correspondence between the runs in PS and the computations in $swarm(PS)$.*

While particle swarm optimization is usually defined over a continuous space, we have transformed discrete versions of particle swarms into graph-transformational swarms because of the following reasons.

1. In the framework of graph transformation, the usual underlying structures are finite graphs or infinite discrete graph in exceptional cases. But all the concepts employed in the paper work for arbitrary sets of nodes and edges including the set of real numbers, the Euclidean space of some dimension or other continuous domains. Nevertheless, we have decided to consider a discrete version of particle swarm optimization as we want to demonstrate the potential of the usual graph transformation rather than to introduce a new kind of graph transformation. Nevertheless, the latter may be an interesting topic of future research.
2. Moreover, implementations of particle swarm models are always discretized. As long as the abstract models are continuous, testing is the only way to validate an implementation against the model. A discrete abstract model between a continuous model and the implementation may allow to prove general properties and to improve the trustworthiness of system development in this way.
3. In the literature, one encounters applications of particle swarm optimization to solve discrete problems (see, e.g., [30–33]). In such a case, a discrete abstract model seems to be appropriate. The particles correspond to problem solutions and the velocity and position updates, as introduced above, are redefined to be applicable to the discrete space. The graph-transformational model $swarm(PS)$ above 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

7. Conclusions

In this paper, we have introduced a graph-transformational approach to swarm computation providing formal methods for the modeling of swarms and the analysis of their correctness and efficiency. The concept exploits graph transformation units and the massive parallelism of rule applications.

As a first example, an ant colony with a simple pheromone-driven cooperation is modeled to illustrate the basic features of graph-transformational swarms. Our main results show that two other major approaches to swarm computation, cellular automata and particle swarms, can be embedded into the graph-transformational framework in a natural way.

The aim of this paper has been to advocate the syntactic and semantic concepts of graph-transformational swarms as a unifying framework for swarm modeling and analysis. To shed more light on the significance and usefulness of our approach, it would be of great interest to demonstrate that it does not only work on the abstract conceptual level, but also on the level of concrete real-world applications. To deliver convincing examples of this kind, quite some further work is needed and is a matter of future research. As first small steps in this direction, we refer to three papers where we consider potential applications concerning the solution of practical problems in cloud-based engineering systems [34] and in dynamic logistic networks with decentralized processing and control in [35] as well as of the routing problem of the automated guided vehicles in [36].

Future studies should provide further correct transformations from models with massive parallelism like ant colony optimization with more sophisticated pheromone-based computation, L-systems and DNA-computing into graph-transformational swarms. The hope is that graph-transformational swarms can serve as a common formal framework for a wide spectrum of swarm approaches.

Author Contributions: The underlying graph-transformational conceptualization and methodology of communities of autonomous units was introduced by H.-J.K. and S.K., the modification to swarms by L.A. The ant colony example is contributed by S.K., the modeling and formal analysis of cellular automata by H.-J.K., and that of particle swarms by L.A. He also designed all the illustrations. H.-J.K. proposed the structure of the paper, its writing was team work. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing not applicable.

Acknowledgments: We would like to thank Aaron Lye as well as all anonymous reviewers for valuable comments and suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bonabeau, E.; Dorigo, M.; Theraulaz, G. *Swarm Intelligence: From Natural to Artificial Systems*; Oxford University Press: Oxford, UK, 1999.
2. Kennedy, J.; Eberhart, R.C. *Swarm Intelligence*; Evolutionary Computation Series; Morgan Kaufman: San Francisco, CA, USA, 2001.
3. Olariu, S.; Zomaya, A.Y. *Handbook of Bioinspired Algorithms and Applications*; Chapman & Hall/CRC: London, UK 2005.
4. Engelbrecht, A.P. *Fundamentals of Computational Swarm Intelligence*; John Wiley & Sons: Pasadena, CA, USA, 2006.
5. Blum, C.; Merkle, D. (Eds.) *Swarm Intelligence: Introduction and Applications*; Natural Computing Series; Springer: New York, NY, USA, 2008.
6. Chakraborty, A.; Kar, A.K. Swarm Intelligence: A Review of Algorithms. In *Nature-Inspired Computing and Optimization: Theory and Applications*; Patnaik, S., Yang, X.S., Nakamatsu, K., Eds.; Modeling and Optimization in Science and Technologies; Springer International Publishing: Cham, Switzerland, 2017; pp. 475–494.
7. Rosenberg, L.; Willcox, G. Artificial Swarm Intelligence. In *Intelligent Systems and Applications, Proceedings of the 2019 Intelligent Systems Conference (IntelliSys 2019), London, UK, 5–6 September 2019*; Bi, Y., Bhatia, R., Kapoor, S., Eds.; Advances in Intelligent Systems and Computing; Springer: Berlin/Heidelberg, Germany, 2019; Volume 1037, pp. 1054–1070.
8. Bansal, J.C.; Singh, P.K.; Pal, N.R. (Eds.) *Evolutionary and Swarm Intelligence Algorithms*; Studies in Computational Intelligence; Springer International Publishing: Cham, Switzerland, 2019.
9. Osaba, E.; Yang, X.S. (Eds.) *Applied Optimization and Swarm Intelligence*; Springer Tracts in Nature-Inspired Computing; Springer: Singapore, 2021.
10. Kreowski, H.J.; Kuske, S. Graph Transformation Units with Interleaving Semantics. *Formal Asp. Comput.* **1999**, *11*, 690–723. [CrossRef]
11. Kuske, S.; Luderer, M. Autonomous Units for Solving the Capacitated Vehicle Routing Problem Based on Ant Colony Optimization. *Electron. Commun. EASST* **2010**, *26*, 23.
12. Kuske, S.; Luderer, M.; Tönnies, H. Autonomous Units for Solving the Traveling Salesperson Problem Based on Ant Colony Optimization. In *Dynamics in Logistics*; Kreowski, H.J., Scholz-Reiter, B., Thoben, K.D., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 289–298.
13. Abdenebaoui, L.; Kreowski, H.J.; Kuske, S. Graph-transformational swarms. In *Proceedings of the Fifth Workshop on Non-Classical Models for Automata and Applications (NCMA 2013), Umeå, Sweden, 13–14 August 2013*; Bensch, S., Drewes, F., Freund, R., Otto, F., Eds.; Österreichische Computer Gesellschaft: Vienna, Austria 2013; pp. 35–50.
14. Abdenebaoui, L. Graph-Transfromational Swarms: A Graph-Transformational Approach to Swarm Computation. Ph.D. Thesis, University of Bremen, Bremen, Germany, 2016.
15. Rozenberg, G. (Ed.) *Handbook of Graph Grammars and Computing by Graph Transformation*; Volume 1: Foundations; World Scientific: Singapore, 1997.
16. Ehrig, H.; Ehrig, K.; Prange, U.; Taentzer, G. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*; Springer: Berlin/Heidelberg, Germany, 2006.
17. Kreowski, H.J.; Klempien-Hinrichs, R.; Kuske, S. Some Essentials of Graph Transformation. In *Recent Advances in Formal Languages and Applications*; Esik, Z., Martin-Vide, C., Mitrana, V., Eds.; Studies in Computational Intelligence; Springer: Berlin/Heidelberg, Germany, 2006; Volume 25, pp. 229–254.
18. Pedemonte, M.; Nesmachnow, S.; Cancela, H. A survey on parallel ant colony optimization. *Appl. Soft Comput.* **2011**, *11*, 5181–5197. [CrossRef]
19. Geiß, R.; Kroll, M. GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In *Proceedings of the 3rd International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*; Schürr, A., Nagl, M., Zündorf, A., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany; Volume 5088, pp. 568–569.
20. Von Neumann, J. The general and logical theory of automata. In *Cerebral Mechanisms in Behavior-The Hixon Symposium, 1948*; Wiley: Pasadena, CA, USA, 1951; pp. 1–41.
21. Von Neumann, J. *Theory of Self-Reproducing Automata*; Burks, A.W., Ed.; University of Illinois Press: Urbana, IL, USA, 1966.
22. Codd, E.F. *Cellular Automata*; Academic Press: New York, NY, USA, 1968.
23. Wolfram, S. *A New Kind of Science*; Wolfram Media Inc.: Champaign, IL, USA 2002.
24. Kari, J. Theory of Cellular Automata: A Survey. *Theor. Comput. Sci.* **2005**, *334*, 3–33. [CrossRef]

25. Peitgen, H.O.; Jürgens, H.; Saupe, D. *Chaos and Fractals: New Frontiers of Science*; Springer: Berlin/Heidelberg, Germany, 1992; Chapter 5, pp. 229–296.
26. Kennedy, J.; Eberhart, R. Particle Swarm Optimization. In Proceedings of the IEEE International Conference on Neural Networks (ICNN'95), Perth, WA, USA, 27 November–1 December 1995; Volume 4, pp. 1942–1948.
27. Poli, R.; Kennedy, J.; Blackwell, T. Particle swarm optimization—An Overview. *Swarm Intell.* **2007**, *1*, 33–57. [CrossRef]
28. Poli, R. Analysis of the publications on the applications of particle swarm optimisation. *J. Artif. Evol. Appl.* **2008**, *2008*, 685175. [CrossRef]
29. Sibalija, T.V. Particle Swarm Optimisation in Designing Parameters of Manufacturing Processes: A Review (2008–2018). *Appl. Soft Comput.* **2019**, *84*, 105743. [CrossRef]
30. Houssein, E.H.; Gad, A.G.; Hussain, K.; Suganthan, P.N. Major Advances in Particle Swarm Optimization: Theory, Analysis, and Application. *Swarm Evol. Comput.* **2021**, *63*, 100868. [CrossRef]
31. Clerc, M. Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem. In *New Optimization Techniques in Engineering*; Studies in Fuzziness and Soft Computing; Springer: Berlin/Heidelberg, Germany, 2004; Volume 141, pp. 219–239.
32. Moraglio, A.; Togelius, J. Geometric particle swarm optimization for the sudoku puzzle. In *GECCO*; Lipson, H., Ed.; ACM: New York, NY, USA, 2007; pp. 118–125.
33. Farmahini-Farahani, A.; Vakili, S.; Fakhraie, S.M.; Safari, S.; Lucas, C. Parallel Scalable Hardware Implementation of Asynchronous Discrete Particle Swarm Optimization. *Eng. Appl. Artif. Intell.* **2010**, *23*, 177–187. [CrossRef]
34. Abdenebaoui, L.; Kreowski, H.J.; Kuske, S. Graph-Transformational Swarms with Stationary Members. In *Proceedings of the 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, 13–15 April 2015*; Camarinha-Matos, M.L., Baldissera, A.T., Di Orio, G., Marques, F., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 137–144.
35. Abdenebaoui, L.; Kreowski, H.J. Modeling of Decentralized Processes in Dynamic Logistic Networks by Means of Graph-Transformational Swarms. *Logist. Res.* **2016**, *9*, 1–13. [CrossRef]
36. Abdenebaoui, L.; Kreowski, H.J. Decentralized Routing of Automated Guided Vehicles by Means of Graph-Transformational Swarms. In *Dynamics in Logistics, Proceedings of the 5th International Conference LDIC, Bremen, Germany, 2016*; Freitag, M., Kotzab, H., Pannek, J., Eds.; Lecture Notes in Logistics; Springer: Berlin/Heidelberg, Germany, 2016.

Short Biography of Authors

Larbi Abdenebaoui is a postdoc researcher in the field of interactive systems with the focus on digital media processing and graph-based computation. Having a background in machine learning, he is exploring the combination of state-of-the-art methods of deep learning with swarm-inspired paradigms in order to perform content-based analysis, modelling and retrieval of image collections. The gained knowledge from this research is shared with students in lectures at the University of Oldenburg.

Hans-Jörg Kreowski is a retired professor for Theoretical Computer Science at the University of Bremen. His main research interests are in the areas of graph transformation and rule-based systems including Petri nets, DNA computing and reaction systems. His research concerns the theoretical foundation, the use of the approaches as modeling and analysis frameworks and the application in Software Engineering, Computer Graphics, Artificial Intelligence, and Logistics. Moreover, he spends some of his efforts to topics in Computer and Society. Readers are refered to www.informatik.uni-bremen.de/theorie (accessed on 06 April 2021) for more information.

Sabine Kuske is a lecturer and research assistant at the university of Bremen in Germany. She is interested in rule based graph transformation, Petri nets and graph algorithms. Readers can contact her at kuske@uni-bremen.de or visit www.informatik.uni-bremen.de/~kuske (accessed on 06 April 2021).