

# Graph-Transformational Swarms with Stationary Members

Larbi Abdenebaoui<sup>(✉)</sup>, Hans-Jörg Kreowski, and Sabine Kuske

University of Bremen, P.O. Box 330440, D-28334 Bremen, Germany  
{larbi, kreo, kuske}@informatik.uni-bremen.de

**Abstract.** The concept of graph-transformational swarms is a novel approach that offers a rule-based framework to model discrete swarm methods. This paper continues the research on graph-transformational swarms by focusing on a special type of members called stationary members. The stationary members are assigned to particular subgraphs of the considered environment graphs. Every stationary member is responsible for calculations and transformations on the assigned area, and the applicability of the member's rules depends only on this area and not on the whole environment. A further advantage of stationary members is that it is easier to guarantee that they can act in parallel than for moving members. Cloud computing 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 forms the cloud as stationary members. We illustrate the proposed concept by means of a case study.

**Keywords:** Swarm computation · Graph transformation · Stationary members · Cloud computing

## 1 Introduction

Swarms in the nature are fascinating phenomena that have inspired various concepts and algorithms in computer science (see, e.g., [3, 4]). However, there seems to be no framework unifying those concepts. Graph-transformational swarms have been proposed in [1] to partly fill this gap by providing a framework to model a variety of discrete swarm methods.

A graph-transformational swarm consists of members that act and interact simultaneously in an environment, which is represented by a graph. The members are all of the same kind or of different kinds. Kinds and members are modeled as graph transformation units [7] each consisting of a set of graph transformation rules specifying the capability of members and a control condition which regulates the application of rules. The basic framework is introduced in [1], where a simple ant colony, cellular automata and discrete particle systems are modeled to demonstrate the usefulness and flexibility of the approach.

This paper continues the research on graph-transformational swarms by focusing on 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.

Graph-transformational swarms are related to other graph transformation approaches to parallelism and distribution as they are surveyed in [9] (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 closest approaches seems to be the graph relabeling systems (see, e.g., [10,11]) where all node labels are changed simultaneously in every step implying massive parallelism and stationarity with respect to nodes.

The paper is organized as follows. Section 2 sketches how cloud-based systems can benefit from the concept of stationary members. In Section 3 we define the new notion of stationary members recalling for this purposes both the basic concepts of the underlying graph transformation approach and the notion of graph-transformational swarms. Section 4 demonstrates the notion of stationary members by means of a case study. The paper ends with the conclusion.

## 2 Contribution to Cloud-Based Engineering Systems

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 via 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 paper 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 follow in Sections 3 and 4.

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 illustrated by the example introduced in Section 4: a cycle freeness test. The proposed solution can be directly applied to a cloud system for detecting deadlocks.

### 3 Graph-Transformational Swarms and Stationary Members

In this section, we define the new notion of stationary members. To achieve this, we recall the concept of graph-transformational swarms starting with the basic components of the chosen graph transformation approach as far as needed in this paper (for more details, see, e.g. [2, 6, 7, 8]).

#### 3.1 Basic Concepts of Graph Transformation

A (*directed edge-labeled*) graph consists of a set of unlabeled nodes and a set of labeled or unlabeled edges such that every edge is directed. If the target is equal to the source, then the edge is called a *loop*. A *match* of a graph  $G$  in a graph  $H$  is an image of  $G$  in  $H$  under a graph morphism. If  $G$  is a subgraph of  $H$ , we use the notation  $G \subseteq H$ .

A rule  $r = (N, L, K, R)$  consists of four graphs: the *negative context*  $N$ , the *left-hand side*  $L$ , the *gluing graph*  $K$ , and the *right-hand side*  $R$  such that  $N \supseteq L \supseteq K \subseteq R$ . We depict a rule as  $N \rightarrow R$  dashing the elements in  $N$  that do not belong to  $L$  and using different shapes for nodes so that  $K$  can be identified as the identical parts of  $L$  and  $R$ . Given the rules  $r_i = (N_i, L_i, K_i, R_i)$  for  $i = 1, \dots, n$ , the *parallel rule*  $p = \sum_{i=1}^n r_i$  is given by the disjoint unions of the respective components of each  $r_i$ .

The application of a rule  $r = (N, L, K, R)$  to a graph  $G$  replaces a match of  $L$  in  $G$  by  $R$  such that the match of  $K$  is kept. If  $L$  is a proper subset of  $N$  the match of  $L$  must not be extendable to a match of  $N$ . A rule application is denoted by  $G \xRightarrow[r]{}$   $H$  where  $H$  is the resulting graph and called a *direct derivation* from  $G$  to  $H$ . A sequence  $G = G_0 \xRightarrow[r_1]{}$   $G_1 \xRightarrow[r_2]{}$   $\dots \xRightarrow[r_n]{}$   $G_n = H$  is called a *derivation* from  $G$  to  $H$ . Such a derivation can also be denoted by  $G \xRightarrow[*]{}$   $H$ . Two direct derivations  $G \xRightarrow[r]{}$   $H_1$  and  $G \xRightarrow[r']{}$   $H_2$  of two rules  $r$  and  $r'$  are (*parallel*) *independent* if the corresponding matches intersect only in gluing items.

The following considerations are based on a parallelization theorem in [5]:

A parallel rule  $p = \sum_{i=1}^n r_i$  can be applied to  $G$  if and only if the rules  $r_i$  for  $i = 1, \dots, n$  can be applied to  $G$  and the matches are pairwise independent. Moreover the  $r_i$  can be applied one after the other in arbitrary order deriving in each case the same graph as the application of  $p$  to  $G$ .

This allows the use of massive parallelism in the context of graph transformation based on local matches of component rules which are much easier to find than matches of parallel rules.

A *control condition*  $C$  is defined over a finite set  $P$  of rules and specifies a set  $SEM(C)$  of derivations. Typical control conditions are priorities and regular expressions over  $P$ . Other control conditions are the expressions  $\|r\|$  and  $[r]$ . The first ex-

pression requires that a maximum number of the rule  $r$  be applied in parallel and the second one requires that the rule  $r$  may be applied or not.

A *graph class expression*  $X$  specifies a set of graphs denoted by  $SEM(X)$ . We use the graph class expression  $id\text{-looped}(G)$  which adds to each node of the graph  $G$  a loop labeled with the name of the node. Another graph class expression is *forbidden*( $H$ ) for a graph  $H$ .  $SEM(\text{forbidden}(H))$  contains all graphs without a subgraph isomorphic to  $H$ .

A *graph transformation unit* is a pair  $gtu = (P, C)$  where  $P$  is a set of rules, and  $C$  is a *control condition* over  $P$ . The *semantics* of  $gtu$  consists of all derivations of the rules in  $P$  allowed by  $C$ . A unit  $gtu$  is *related* to a unit  $gtu_0$  if  $gtu$  is obtained from  $gtu_0$  by relabeling. The set of units related to  $gtu_0$  is denoted by  $RU(gt u_0)$ .

### 3.2 Graph-Transformational Swarms

A graph-transformational swarm consists of members of the same kind or of different kinds to distinguish between different roles members can play. All members act simultaneously in a common environment represented by a graph. The number of members of each kind is given by the size of the kind. While a kind is a graph transformation unit, the members of this kind are modeled as units related to the kind so 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. The choice of rules depends on their applicability and the control conditions of the members. We allow to provide a swarm with a cooperation condition. Moreover, 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 1 (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, called *kinds*,  $size$  associates a *size*  $size(k) \in \mathbb{N}_{>0}$  with each kind  $k \in K$ ,  $M$  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 called *cooperation condition*, and  $goal$  is a graph class expression specifying the *goal*<sup>1</sup>.

A swarm may be represented schematically depicting the components *initial*, *kinds*, *size*, *members*, *cooperation* and *goal* followed by their respective values.

**Definition 2 (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.

---

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

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  $(\emptyset, \emptyset, \emptyset, \emptyset)$  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 within a parallel step.

### 3.3 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. More formally, we get the following definition.

**Definition 3 (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  $G(m) \in SUB$ .

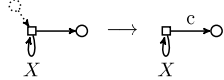
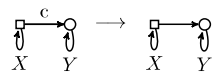
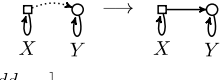
- (3) Each left-hand side of each rule of each member  $m$  contains a subgraph that matches only in  $G(m)$  and the rest of the match can be found in the neighborhood of  $G(m)$ .

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

## 4 A Case Study: Cycle Freeness Test

In this section, we illustrate the notion of stationary members by means of a simple and well-known decision problem. We provide a swarm 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. We have chosen this simple example because more sophisticated examples would need too much space. But to illustrate all the features of swarms, we solve the cycle freeness problem in a dynamic setting meaning that new edges can be added to the underlying graph from time to time so that after enough edge additions every initial graph ends up with cycles.

The swarm that tests a simple, unlabeled and directed graph  $G$  for cycle freeness is depicted below. It gets the graph  $G$  as a parameter and we assume 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  $mark_X$  which marks an edge outgoing of a node with an  $X$ -loop provided that there is no incoming unlabeled edge. The control condition  $\|mark_X\|$  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 a single rule that turns a marked edge into an unmarked one. The  $X$ - and  $Y$ -loop identify source and 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  $[add_{X,Y}]$  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* and this again 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.

<p><math>\text{cyclefree}(G)</math></p> <p>initial: <math>\text{id-looped}(G)</math></p> <p>kinds: <math>\text{marker}, \text{resetter}, \text{adder}</math></p> <p>size: <math>n, n^2-n, n^2-n</math></p> <p>members: <math>\text{marker}_i</math> for <math>i \in [n]</math>,  <math>\text{resetter}_{i,j}</math> for <math>i, j \in [n], i \neq j</math>,  <math>\text{adder}_{i,j}</math> for <math>i, j \in [n], i \neq j</math></p> <p>coop: <math>\text{marker}!; (\text{resetter}, \text{adder}, \text{marker}!)*</math></p> <p>goal: <math>\text{forbidden}(\text{O} \rightarrow \text{O})</math></p>	<p>marker</p> <p>rules:</p> <p><math>\text{mark}_X</math>: </p> <p>control: <math>\ \text{mark}_X\ </math></p>
<p>resetter</p> <p>rules:</p> <p><math>\text{reset}_{X,Y}</math>: </p>	<p>adder</p> <p>rules:</p> <p><math>\text{add}_{X,Y}</math>: </p> <p>control: <math>[\text{add}_{X,Y}]</math></p>

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  $\text{mark}_i$  is fixed by the unique *i*-loop and varies only in the outgoing edges. Two matches of  $\text{mark}_i$  are independent if they access different outgoing edges which can be checked locally. Matches of  $\text{mark}_i$  and  $\text{mark}_j$  for  $i \neq j$  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 1.** The swarm  $\text{cyclefree}(G)$  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.

## 5 Conclusion

In this paper, we have introduced the notion of graph-transformational swarms with stationary members, which establishes a connection between different areas of research: graph transformation, swarm computation and cloud computing. The idea is to model a cloud as a graph and to apply the concepts and results of graph-transformational swarms within the framework of cloud computing.

We have presented a case study that demonstrates how to apply the notion of stationary members to solve graph problems. The case study shows 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 in general NP-hard problems. Their solutions can profit from a combination of swarm heuristics and the massive parallelism within the proposed framework.

**Acknowledgement.** We are grateful to the anonymous reviewers for their valuable comments.

## References

1. Abdenebaoui, L., Kreowski, H.-J., Kuske, S.: Graph-transformational swarms. In: Bensch, S., Drewes, F., Freund, R., Otto, F. (eds.) Proceedings of the Fifth Workshop on Non-Classical Models for Automata and Applications – NCMA 2013, Umea, Sweden, August 13–August 14, pp. 35–50. Österreichische Computer Gesellschaft (2013)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
3. Engelbrecht, A.P.: Fundamentals of Computational Swarm Intelligence. John Wiley & Sons (2006)
4. Kennedy, J., Eberhart, R.C.: Swarm Intelligence. Evolutionary Computation Series. Morgan Kaufman, San Francisco (2001)
5. Kreowski, H.-J.: Manipulationen von Graphmanipulationen. PhD thesis, Technische Universität Berlin (1977)
6. Kreowski, H.-J., Klempien-Hinrichs, R., Kuske, S.: Some essentials of graph transformation. In: Esik, Z., Martin-Vide, C., Mitrana, V. (eds.) Recent Advances in Formal Languages and Applications. SCI, vol. 25, pp. 229–254. Springer, Heidelberg (2006)
7. Kreowski, H.-J., Kuske, S., Rozenberg, G.: Graph transformation units – an overview. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 57–75. Springer, Heidelberg (2008)
8. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific, Singapore (1997)
9. Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution, vol. 3. World Scientific (1999)
10. Métivier, Y., Sopena, E.: Graph Relabelling Systems: A General Overview. Computers and Artificial Intelligence **16**(2) (1997)
11. Bauderon, M., Métivier, Y., Mosbah, M., Sellami, A.: Graph Relabelling Systems: A Tool for Encoding, Proving, Studying and Visualizing Distributed Algorithms. Electronic Notes in Theoretical Computer Science **51**, 93–107 (2002)