Autonomous Units for Solving the Capacitated Vehicle Routing Problem Based on Ant Colony Optimization [*]

Sabine Kuske, Melanie Luderer, and Hauke Tönnies

Abstract. Communities of autonomous units and ant colony systems have fundamental features in common. Both consists of a set of autonomously acting units that transform and move around a common environment that is usually a graph. In contrast to ant colony systems, the actions of autonomous units are specified by graph transformation rules which have a precisely defined operational semantics and can be visualized in a straighforward way. In this paper, we model an ant colony system solving the capacitated vehicle routing problem as a community of autonomous units. The presented case study shows that the main characteristics such as tour construction and pheromone update can be captured in a natural way by autonomous units. Hence, autonomous units provide a formal and visual framework for ant colony optimization algorithms.

1 Introduction

Communities of autonomous units are rule-based systems, in which the units act and interact autonomously in a common environment while striving for a goal (cf. [KK07,KK08,HKK09]). More concretely, every autonomous unit is composed of a set of graph transformation rules, a control condition, a specification of initial private states, and a goal. Moreover, it can ask auxiliary units for help. Autonomous units transform the common environment and their private states simultaneously, can communicate with each other via the common environment, and may act in parallel. A current state of an autonomous unit consists of a common environment and a private state which are both graphs. An autonomous unit specifies all state transformation processes that (1) start with an initial private state and an arbitrary common environment (2) are allowed by the control condition, and (3) can be obtained via (parallel) applications of the unit's rules, auxiliary units, and other autonomous units in the community. Hence, the semantics of a single autonomous unit includes actions of other autonomous units which are not known by the unit. This means that the semantics of an autonomous unit is loose in the sense that it is defined with respect to a set of (parallel) rules that model the actions of other units. These rules are called metarules. A state transformation process is called successful if it meets the goal.

[*] The authors would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

A community consists of a set of autonomous units, a specification of initial common environments, a global control condition, and an overall goal. A current state of a community is composed of a current common environment plus a private state for every autonomous unit. The semantics of a community consists of all state transformation processes performed by the autonomous units, allowed by the global control condition, and starting with an initial state. A transformation process is successful if it reaches the overall goal. The basic components of communities are provided by a graph transformation approach consisting of a class of graphs, a class of graph class expressions, a class of rules with a rule application operator, and a class of control conditions. In the literature there exists a variety of graph transformation approaches that differ mainly in the kind of rules and graphs (cf. [Roz97] for an overview on graph transformation approaches). They all can be used as underlying approach for communities.

Ant colony systems consist of a set of autonomously behaving artificial ants that move around a common graph and make their decisions according to the pheromone concentration in their neighbourhood. They are inspired by the way how ants find short routes between food and their formicary and have been shown to be well-suited not only for the solving of shortest path problems, but for a series of more complex problems, typically ocurring in logistics (cf. [DS04]). Basically, in an ant colony system, a set of ants constructs solutions for a given problem (mostly NP-hard) by moving along the edges of an underlying graph. According to the quality of the constructed solutions the ants walk back and put some pheromone on the traversed items, i.e., the better the solution is the more pheromone is placed by an ant. During solution construction the pheromone concentration as well as some further heuristic value help the ants to decide where to go in each step. Every ant has a memory for storing important information such as the length of the traversed path, etc.

In this paper, we show that ant colony systems can be modeled by communities of autonomous units in a natural way. This is illustrated with the example of the Capacitated Vehicle Routing Problem (CVRP) (cf., e.g., [RDH04,DS04]). The advantages of modeling ant colony systems as communities of autonomous units are the following. (1) Autonomous units provide ant colony systems with a well-founded operational semantics so that verification techniques for graph transformation can be applied to ant colony systems. (2) The fact that ant actions can be specified as graph transformation rules allows for a visual modeling of ant algorithms and hence for a visual representation of ant colony behaviour. (3) Existing graph transformation tools such as GrGEN [GK08] or AGG [ERT99] can be used to implement ant algorithms.

This paper is organized as follows. In Section 2, ant colony systems for the heuristic solving of optimization problems are briefly introduced and a particular ant colony optimization algorithm for solving the CVRP is recalled. Section 3 presents a particular graph transformation approach that is used throughout this paper. Section 4 introduces autonomous units and communities of autonomous units. Section 5 shows how fundamental features of ant colony systems can be modeled with autonomous units by translating an ant colony system solving the CVRP into a community. The conclusion is given in Section 6.

2 Ant Colony Opimization

Ant colony optimization (ACO) systems are algorithmic frameworks for the heuristic solving of optimization problems, typically problems belonging to the complexity class NP-hard, since no efficient algorithms for this kind of problems are known that always solve the problem. The idea of ACO originates in the observation of how ants find short ways between food and their formicary. An individual ant can hardly see and has a very narrow perspective of its environment. While searching for food, it leaves a chemical substance on the ground, called pheromone, which can be sensed by other ants and influence their route decision. The higher the concentration of pheromone along a way, the higher the probability that an ant will choose this way as well, thus leaving even more pheromone. The crucial point is that pheromone evaporates with time. An ant following a short route to food will return sooner to the formicary (returning obviously on the same way) so that the pheromone concentration on shorter routes becomes more intense than on longer routes. The higher pheromone concentration makes more ants choose the short route which in turn raises the pheromone concentration further. Finally, almost all ants end up choosing one short route, although not necessarily the shortest one. Since typical optimization problems can be nicely modeled as graphs, it is the prefered data structure for ACO. The graphs used in this paper are edge-labeled and undirected and can be defined as follows.

Definition 1 (Graphs). A *graph* is a tuple (V, E, att, m), where V is a finite set of *nodes*, E is a finite set of *edges* such that V and E are disjoint, $att : E \to \bigcup_{k\in\{1,2\}} \binom{V}{k}$ assigns to every edge a set of one or two *sources* in V, and m is a mapping that assigns a *label* to every edge in E. A graph with no nodes and no edges is called the *empty graph* which is denoted by $\emptyset$. The components of G are also denoted by V_G, E_G, respectively. The set of all graphs is denoted by $\mathcal{G}$.

A solution to an optimization problem consists typically of a tour (e.g. an ordered sequence of nodes) within the given graph. Intuitively, the complexity of most NP-hard optimization problems lies in the exponentially growing number of possible tours when new nodes and edges are added. The lack of an efficient search method for the 'best' way requires an (almost) exhaustive search of all the possible tours. To solve an optimization problem with ACO, some additional information is needed. We define optimization problems as follows.

Definition 2 (Optimization Problem). An optimization problem is a 6-tuple $(CG, d, \tau, \eta, S, g)$ where $CG \in \mathcal{G}$ is a *construction graph*, d is a function that associates every edge with a cost value (e.g. the distance), τ is a function that associates every edge with a pheromone value, η is a function that associates every edge with a number as an heuristic value for the quality of the edge, $S \subseteq V^*$ is the set of *solutions*, and g assigns a *cost* $g(s)$ to every $s \in S$.

Basically, ACO works as follows. At first, a predefined number of ants are placed randomly at some nodes. These ants decide in parallel which edge they follow in the next step according to a transition rule. Let a be an index to choose one of n ants and U_a the set of all edges that can be chosen from ant a residing at some node. The decision, which edge $e \in U_a$ to take, is probability-based. The probabilities are calculated as follows.

$$p_a(e) = \frac{[\tau(e)]^\alpha \cdot [\eta(e)]^\beta}{\sum_{e \in U_a} [\tau(e)]^\alpha \cdot [\eta(e)]^\beta} \quad \forall e \in U_a$$

In words this formula states that ants prefer edges with low cost und a high concentration of pheromone. The experimental parameters α and β control the influence of the pheromone resp. heuristic value in the decision. In every step this formula is applied, until all the ants have constructed a complete tour.

The next step concerns the pheromone values. Simulating the evaporation, the values of τ are reduced: $\tau(e) \leftarrow (1-\rho) \cdot \tau(e) \quad \forall e \in E_{CG}$ where ρ is a pheromone decay parameter in the intervall $(0, 1]$. Furthermore the release of pheromone of the ants is simulated:

$$\tau(e) \leftarrow \tau(e) + \sum_{a=1}^{n} \Delta\tau_a(e), \text{ with } \Delta\tau_a(e) = \begin{cases} \frac{1}{length(tour_a)} & , e \in tour_a \\ 0 & otherwise \end{cases}$$

where $tour_a$ is the solution constructed by ant a. In contrast to nature, the release of pheromone takes place after the ants constructed a complete tour, since the amount of pheromone corresponds to the overall quality of the tour (e.g. the length of the tour). Furthermore, in some ACO systems not every ant leaves pheromone, but just the ones having constructed the best tours.

Now the ants are placed again at some randomly chosen nodes and the algorithm starts with the modified values of pheromone. Some variants of this basic ACO yielding better performance have been proposed in the literature. Details can be found in [DS04].

2.1 Application: Capacitated Vehicle Routing Problem

An important application field of ACO concerns all kinds of tour planning with the Traveling Salesperson Problem (TSP) as the most famous one. Another problem often occurring in distribution logistics is the so called Capacitated Vehicle Routing Problem (CVRP), which can be described as follows. A number of customers must be served with some goods that are stored at a central depot. A number of vehicles with finite and equal capacity is available. The aim is to find a set of tours such that the demands of all customers are met and the total cost (the sum of the distances of the tours) is minimized. Combinatorially, a solution can be formally described as a partition of the cities into m routes $\{R_1, \ldots, R_m\}$. Each route must satisfy the condition $\sum_{j \in R_i} dem_j \leq k$, where dem_j describes the demand of the j-th customer and k is the capacity restriction of the vehicles. Within each partition, an associated permutation function specifies the customer order.

Relaxing the conditions by allowing any partition (respectively setting $k = \infty$), the CVRP is transformed into an instance of the Multiple Traveling Salesperson Problem. Leaving the condition unchanged but with a cost function that counts the number of partitions CVRP becomes the well-known bin packing problem. CVRP contains in this sense two NP-hard problems, which in practice makes it a lot more complicated to solve than TSP for example and it seems a good idea to use ACO. A formulation of CVRP according to Definition 2 is quickly found. Nevertheless, there are different ways to design the function $\eta : E_{CG} \to \mathbb{R}$. One easy possibility consists of the reciprocal cost-value of the edge.

Nevertheless, sometimes other methods are used to calculate the heuristic values; one elegant way is based on the so-called *Savings algorithm*. Starting from the initial (and unfavoured) solution, where every route consists of exactly one customer, it is calculated, how the quality of the solution changes (how much one would save), putting two customers i and j in one route. Let d_{i0} denote the distance between customer i and the depot and d_{ij} the distance between customer i and j. Then the saving value obtained by merging the routes R_i and R_j together is calculated as follows:

$$\begin{aligned} s_{ij} &= 2 * d_{i0} + 2 * d_{j0} - (d_{i0} + d_{ij} + d_{j0}) && (1) \\ &= d_{i0} + d_{j0} - d_{ij} && (2) \end{aligned}$$

Elaborated experiments concerning the performance of ACO and Saving Algorithm for the CVRP can be found in [RDH04].

3 A Graph Transformation Approach

Graph transformation approaches provide the main ingredients for communities of autonomus units. They consist of a class of graphs, a class of rules, a class of control conditions, and a class of graph class expressions. The graphs are used to represent the common environments and the private states of communities. The rules are needed to transform these graphs. Moreover, control conditions can restrict the non-determinism of rule application, and with graph class expressions one can specify specific graph sets such as initial environments or goals to be reached. In the literature, there exists a series of different graph transformation approaches (cf. [Roz97]).

In the following, we tailor a particular graph transformation approach that can be used for modeling ACO algorithms. Concretely, the rule class and the graph class are based on the double-pushout approach [CEH$^+$97]. Additionally, we introduce a class of control conditions that is suitable for autonomous units running in parallel. These control conditions are proactive meaning that rules must always be applied as soon as possible. The class of graph class expressions allows to specify graph languages in a rule-based way.

3.1 Graphs and Rules

The graphs we use are edge-labeled and undirected as presented in Section 2. Subgraphs and graph morphisms are defined as follows.

Definition 3 (Subgraph, graph morphism). For $G, G' \in \mathcal{G}$, the graph G is a *subgraph* of G', denoted by $G \subseteq G'$, if $V_G \subseteq V_{G'}$, $E_G \subseteq E_{G'}$, $att(e) = att'(e)$, and $m(e) = m'(e)$ for all $e \in E_G$. A *graph morphism* $g\colon G \to G'$ is a pair (g_V, g_E) of mappings with $g_V\colon V_G \to V_{G'}$ and $g_E\colon E_G \to E_{G'}$ such that labels and sources are kept, i.e., for all $e \in E_G$, $g_V(att_G(e)) = att_{G'}(g_E(e))$ and $m_{G'}(g_E(e)) = m_G(e)$.[1] The image of G in G' is the subgraph $g(G)$ of G' such that $V_{g(G)} = g_V(V_G)$ and $E_{g(G)} = g_E(E_G)$.

Remark. In the following, the subscripts V and E of g_V and g_E are often omitted, i.e., $g(x)$ means $g_V(x)$ for $x \in V$ and $g_E(x)$ for $x \in E$.

Graphs are depicted as usual with round or boxed nodes and lines as edges. A loop can be omitted by putting its label inside the node to which the loop is attached. This can be done for at most one loop per node. We assume the existence of a special label *unlabeled* that is omitted in graph drawings.

Graphs can be modified by rules consisting of a negative context, a left-hand side, a gluing graph, and a right-hand side. Roughly speaking, the negative context specifies components that must not occur in the graph to which the rule is applied. The left-hand side, the gluing graph, and the right-hand side are used to determine which components should be deleted, kept and added, respectively. In every computation step of a community, the autonomous units transform the common environment and their private states simultaneously. For this purpose, every unit applies pairs of rules (r_1, r_2), where the first rule r_1 is applied to the common environment and r_2 to the private state.

Definition 4 (Rule, rule pair). A *rule* r is a quadruple (N, L, K, R) of graphs with $N \supseteq L \supseteq K \subseteq R$ where N is the *negative context*, L is the *left-hand side*, K is the *gluing graph*, and R is the *right-hand side*. If all components of r are empty, r is the *empty rule*. The set of all rules is denoted by $\mathcal{R}$. A *rule pair* is a pair of rules $r = (r_1, r_2)$ where r_1 is called the *global rule* and r_2 the *private rule*. The set of all rule pairs is denoted by $\widetilde{\mathcal{R}}$.

Remark. A rule pair $r = (r_1, r_2)$ where r_2 is the empty rule can be regarded as a single rule. Hence, in the following, we often do not distinguish between single rules and rule pairs with an empty private rule.

A rule (N, L, K, R) is depicted as $N \to R$ where the nodes and edges of K have the same forms, labels, and relative positions in N and R. The nodes of N that do not belong to L are coloured grey. The edges of N that do not belong to L are dashed. Fig. 1 shows a rule in which the negative context consists of a round node and two rectangle nodes. Each of the rectangle nodes has exactly

[1] For a mapping $f\colon A \to B$ and $C \subseteq A$ the set $f(C)$ is defined as $\{f(x) \mid x \in C\}$, i.e., $g_V(att_G(e)) = \{g_V(v) \mid v \in att_G(e)\}$.

one loop labeled with a and b, respectively. The round node is connected to both rectangle nodes. The left-hand side contains the round node, the a-node (i.e., the rectangle node with the a-loop) and the edge between both. The gluing graph consists of the round node, and the right-hand side is obtained from the gluing graph by connecting the round node with a new b-node.

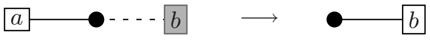


Fig. 1. A rule

A rule pair $r = ((N_1, L_1, K_1, R_1), (N_2, L_2, K_2, R_2))$ (with non-empty private rule) is depicted as $L_1|L_2 \to R_1|R_2$ where the negative contexts and the gluing graphs are represented as in single rules.

A rule (N, L, K, R) is applied to a graph as follows. (1) Choose an image $g(L)$ of L in G. (2) Check if $g(L)$ has no negative context given by N up to L. (3) Delete $g(L)$ up to $g(K)$ from G provided that no dangling edges are produced. (4) Glue R and the remaining graph in K. The construction needed in the fourth step can be defined as follows.

Definition 5 (Gluing of graphs). Let $K \subseteq R$ and $h: K \to Z$. Let $\approx_V$ be the equivalence relation on $V_Z + V_R$ generated by the relation $\{(v, h_V(v)) \mid v \in V_K\}$ and let $\approx_E$ be the equivalence relation on $E_Z + E_R$ generated by $\{(e, h_E(e)) \mid e \in E_K\}$. Let $(V_Z + V_R)/\approx_V$ and $(E_Z + E_R)/\approx_E$ be the quotient sets of the disjoint union $V_Z + V_R$ and $E_Z + E_R$, respectively. Then the *gluing* of Z and R in K with respect to h yields the graph $D = ((V_Z + V_R)/\approx_V, (E_Z + E_R)/\approx_E, att, m)$ where for all $e \in (E_Z + E_R)/\approx_E$

$$att(e) = \begin{cases} [att_Z(\overline{e})] \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_Z \text{ }^2 \\ [att_R(\overline{e})] \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_R - E_K \end{cases}$$

$$m(e) = \begin{cases} m_Z(\overline{e}) \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_Z \\ m_R(\overline{e}) \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_R - E_K \end{cases}$$

The application of a rule to a graph is formally defined as follows.

Definition 6 (Rule application). Let $r = (N, L, K, R) \in \mathcal{R}$, let $G \in \mathcal{G}$, and let $g: L \to G$ such that g is injective and the following *gluing condition* is satisfied.

– If $L \subset N$, there exists no $g': N \to G$ with $g'(x) = g(x)$ for all $x \in V_L \cup E_L$.
– For all $e \in E_G - E_{g(L)}$, $att_G(e) \subseteq V_G - (V_{g(L)} - V_{g(K)})$.

[2] For a quotient set $A/\approx$, $[]: A \to A/\approx$ denotes its natural associated function.

Then r is applied to G by (1) deleting $V_{g(L)} - V_{g(K)}$ and $E_{g(L)} - E_{g(K)}$, and (2) constructing the gluing of the resulting graph D and R in K with respect to $g|K\colon K \to D$ where $g|K(x) = g(x)$ for all $x \in V_K \cup E_K$. The *semantic relation of* r is denoted by $SEM(r)$ and consists of all pairs (G, G') such that G' can be derived from G via the application of r. For a set $P \subseteq \mathcal{R}$, we define $SEM(P) = \bigcup_{r \in P} SEM(r)$. For $(r_1, r_2) \in \widetilde{\mathcal{R}}$, the semantic relation is equal to $\{((G_1, G_2), (G'_1, G'_2)) \mid (G_i, G'_i) \in SEM(r_i), i = 1, 2\}$.

Remark. The described kind of applying graph transformation rules corresponds to the double-pushout approach presented in e.g. [CEH$^+$97], where also non-injective matchings of the left-hand side are allowed.

The rule in Fig. 1 can be applied to a graph containing a node v connected to an a-node but not connected to a b-node. Its application removes the a-node plus the edge to v and adds a b-node and an edge from this b-node to v. Because of the gluing condition, the a-node is only connected to v but not to other nodes; otherwise its deletion would produce dangling edges.

In general, the autonomous units of a community apply their rules in parallel. A parallel rule application step involving two rules can be defined as follows.

Definition 7 (Parallel rule application). Let $G \in \mathcal{G}$ and for $i = 1, 2$, let $r_i = (N_i, L_i, K_i, R_i)$ be two rules. Let $g_i\colon L_i \to G$ be two injective graph morphisms that satisfy the gluing condition of Definition 6 and the *independence condition* $g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$.[3] Then r_1 and r_2 can be applied in parallel to G by (1) deleting $V_{g_i(L)} - V_{g_i(K)}$ and $E_{g_i(L)} - E_{g_i(K)}$ (for $i = 1, 2$), and (2) constructing the gluing of the resulting graph D and $R_1 + R_2$ in $K_1 + K_2$ with respect to $g\colon K_1 + K_2 \to D$, where $g(x) = g_i(x)$ if $x \in V_{K_i} \cup E_{K_i}$, for $i = 1, 2$.[4]

Remarks.

1. Definition 7 can be extended in a straightforward way from two rules to arbitrary non-empty multisets of rules. For a multiset m of rules, $SEM(m)$ denotes the set of all $(G, G') \in \mathcal{G} \times \mathcal{G}$ where G' is derived from G via the parallel application of the rules in m. A multiset m of rules will be called a *parallel rule*, and for a set $P \subseteq \mathcal{R}$, the set of all parallel rules over P is denoted by P_*.
2. For a rule pair $r = (r_1, r_2)$, $SEM(r||m)$ denotes all $((G_1, G_2), (G'_1, G'_2)) \in (\mathcal{G} \times \mathcal{G}) \times (\mathcal{G} \times \mathcal{G})$ where G'_1 is derived from G_1 by applying the multiset obtained from adding r_1 to m, and $(G_2, G'_2) \in SEM(r_2)$.

3.2 Control Conditions

It is often desirable to restrict the non-determinism of rule application. This can be achieved with control conditions. Concretely, we use as control conditions

[3] For $G_1, G_2 \in \mathcal{G}$ the intersection $G_1 \cap G_2$ yields the pair (V, E) where $V = V_{G_1} \cap V_{G_2}$ and $E = E_{G_1} \cap E_{G_2}$. Moreover, we have $(V_1, E_1) \subseteq (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

[4] The morphism g may be non-injective.

regular expressions equipped with *as long as possible* and the parallel operator $||$.

Definition 8 (Control conditions). Let ID be a set such that $P \subseteq ID$ for some set P of rule pairs. Then the class $\mathcal{C}(ID)$ of *control conditions* over ID is inductively defined as follows.

1. $\{lambda\} \cup ID \cup \{x! \mid x \in P\} \subseteq \mathcal{C}(ID)$.
2. For $c, c_1, c_2 \in \mathcal{C}(ID)$, we have $(c_1 + c_2), (c_1\,; c_2), (c^*), (c_1||c_2) \in \mathcal{C}(ID)$.

Remark. For practical applications, the set ID would consist of names refering to rule pairs (or units) but for technical simplicity we do not distinguish between rule pairs (units) and their names.

If ID consists only of rule pairs, a semantics of control conditions can be defined in an intuitive way. Roughly speaking, the condition *lambda* applies no rule. Every rule pair r is a control condition that prescribes one application of r. The condition $c_1 + c_2$ stands for applying c_1 or c_2, $c_1\,; c_2$ means that c_1 must be applied before c_2, c^* applies c arbitrarily often, $r!$ requires that the pair r be applied as long as possible, and $c_1||c_2$ allows only transformations where c_1 and c_2 are applied in parallel. For example, the expression $r_1; r_2^* + r_3!$ allows all sequences in which r_1 is applied before an arbitrarily often application of r_2 or in which r_3 is applied whenever this is possible.[5]

As stated before, the application of a rule pair by an autonomous unit *aut* is generally done in parallel with transformations of the common environment executed by other autonomous units. Moreover, it may also happen that other units perform actions before or after rule applications of *aut*. When defining the semantics of control conditions the rules of other units are not known. Hence, the semantics is loose, i.e., it is defined with respect to a set $\mathcal{MR}$ of parallel rules called *metarules*. For modeling ant-based systems in a suitable way, we require additionally that rules be applied as early as possible. This leads to proactive transformation processes. In more detail, for a set P of rule pairs, every proactive transformation process s specified by a control condition $c \in \mathcal{C}(P)$ must have the following property: If in s an application of a rule pair $r \in P$ is preceded by a sequence of k mere metarule applications, the application of r cannot be shifted to any of the k preceding steps. For defining a proactive semantics of control conditions, i.e., a semantics that consists only of proactive transformation processes, we also have to define proactive transformation processes in which no metarules are applied after the last application of a rule in P. For example, for $i = 1, 2$, let $SEM_{\mathcal{MR}}(r_i)$ be the proactive transformation processes specified by the rule r_i. In order to get all proactive transformation processes specified by $r_1\,; r_2$, we cannot take the sequential composition of the processes in $SEM_{\mathcal{MR}}(r_1)$ and $SEM_{\mathcal{MR}}(r_2)$, because of the following reason. Let s_1 be a transformation process in $SEM_{\mathcal{MR}}(r_1)$ in which some metarules are applied after r_1. Let s_2 be a transformation process in $SEM_{\mathcal{MR}}(r_2)$. Then the sequential composition

[5] The operator * has a stronger binding than ; which in turn has a stronger binding than $+$.

$s_1 \circ s_2$ is not proactive if r_2 is applicable after the application of r_1 in s_1 but before the end of s_1 because in this case the application of r_2 can be shifted to an earlier step. Hence, before sequentially composing s_1 and s_2 we have to cut all metarule applications from s_1 that take place after the application of r_1.

In the following, a proactive semantics of control conditions is defined for the case where *ID* consists of rules, only. In Section 4 we show how this definition can be employed for the more general case where *ID* contains units, too.

Definition 9 (Proactive semantics of control conditions). Let $\mathcal{MR} \subseteq \mathcal{R}_*$ be a set of parallel rules called *metarules* and let $P \subseteq \widetilde{\mathcal{R}}$. Then for each control condition in $\mathcal{C}(P)$ its *proactive semantics* is defined as follows.

1. $SEM_{\mathcal{MR}}(lambda)$ consists of all sequences $(G_0, \ldots, G_n)$ of graph pairs such that for $i = 1, \ldots, n$, $(G_{i-1}, G_i) \in SEM(m)$ for some $m \in \mathcal{MR}$.[6] Moreover, we define $CUT(SEM_{\mathcal{MR}}(lambda)) = \mathcal{G} \times \mathcal{G}$.
2. $SEM_{\mathcal{MR}}(r)$ consists of all sequences $s = (G_0, \ldots, G_n)$ for which there exist some $j \in \{1, \ldots, n\}$ and $m_1, \ldots, m_n \in \mathcal{MR}$ such that for $i = 1, \ldots, j-1$ and $i = j+1, \ldots, n$, $(G_{i-1}, G_i) \in SEM(m_i)$, $(G_{j-1}, G_j) \in SEM(r||m_j)$ and for $i = 0, \ldots, j-1$, there is no $G \in \mathcal{G} \times \mathcal{G}$ such that $(G_i, G) \in SEM(r||m_i)$. Moreover, we define $cut(s) = \text{TRUE}$ iff $j = n$ and $CUT(SEM_{\mathcal{MR}}(r)) = \{s \in SEM_{\mathcal{MR}}(r) \mid cut(s) = \text{TRUE}\}$.
3. $SEM_{\mathcal{MR}}(c_1 + c_2) = SEM_{\mathcal{MR}}(c_1) \cup SEM_{\mathcal{MR}}(c_2)$ and
$$CUT(SEM_{\mathcal{MR}}(c_1 + c_2)) = CUT(SEM_{\mathcal{MR}}(c_1)) \cup CUT(SEM_{\mathcal{MR}}(c_2)).$$
4. $SEM_{\mathcal{MR}}(c_1\,;c_2) = CUT(SEM_{\mathcal{MR}}(c_1)) \circ SEM_{\mathcal{MR}}(c_2)$ and
$$CUT(SEM_{\mathcal{MR}}(c_1\,;c_2)) = CUT(SEM_{\mathcal{MR}}(c_1)) \circ CUT(SEM_{\mathcal{MR}}(c_2)).^{7}$$
5. $SEM_{\mathcal{MR}}(c^*) = SEM_{\mathcal{MR}}(lambda) \cup CUT(SEM_{\mathcal{MR}}(c))^* \circ SEM_{\mathcal{MR}}(c)$. Moreover, $CUT(SEM_{\mathcal{MR}}(c^*)) = (CUT(SEM_{\mathcal{MR}}(c)))^*$.
6. $SEM_{\mathcal{MR}}(r!) = CUT(SEM_{\mathcal{MR}}(r^*)) \circ \{(G_0, \ldots, G_k) \in SEM_{\mathcal{MR}}(lambda) \mid G_i \in red(r) \text{ for } i = 1, \ldots, k\}$ where for $(r_1, r_2) \in \widetilde{\mathcal{R}}$, $red(r_1, r_2)$ consists of all $(G_1, G_2) \in \mathcal{G} \times \mathcal{G}$ such that r_1 is not applicable to G_1 or r_2 is not applicable to G_2. Moreover, $CUT(SEM_{\mathcal{MR}}(r!)) = \{(G_0, \ldots, G_k) \in CUT(SEM_{\mathcal{MR}}(r^*)) \mid G_k \in red(r)\}$.
7. $SEM_{\mathcal{MR}}(c_1||c_2) = SEM_{\mathcal{MR} \cup Rules(c_2)_*}(c_1) \cap SEM_{\mathcal{MR} \cup Rules(c_1)_*}(c_2)$ where for $i = 1, 2$, $Rules(c_i)$ is the set of all rule pairs occurring in c_i. Moreover, $CUT(SEM_{\mathcal{MR}}(c_1||c_2)) = SEM_{\mathcal{MR}}(c_1||c_2) \cap (CUT(SEM_{\mathcal{MR} \cup Rules(c_2)_*}(c_1)) \cup CUT(SEM_{\mathcal{MR} \cup Rules(c_1)_*}(c_2)))$.

Remark. In the above definition $SEM_{\mathcal{MR}}(c)$ is the set of proactive transformation processes specified by c whereas $CUT(SEM_{\mathcal{MR}}(c))$ is needed for defining the proactive semantics of control conditions involving sequential composition.

[6] In this transformation, the second component of every graph pair remains unchanged, because m is a multiset of single rules.

[7] For sets of sequences S, S', their sequential composition is denoted by $S \circ S'$, and S^* is defined as $\bigcup_{i \in \mathbb{N}} S^i$ with $S^0 = \mathcal{G} \times \mathcal{G}$ and $S^{i+1} = S^i \circ S$.

3.3 Graph Class Expressions

In order to use graph transformation in a meaningful way, it should be possible to specify initial and terminal graphs of graph transformation processes with graph class expressions. In general, a graph class expression can be any expression that specifies a set of graphs. In particular, the graph class expressions used in this paper are the following.

Definition 10 (Graph class expressions). The class $\mathcal{X}$ of all graph class expressions is defined as follows.

1. $all, empty, red(P), (P, C) \in \mathcal{X}$ with $P \subseteq \mathcal{R}$ and $C \in \mathcal{C}(P)$ where $SEM(all) = \mathcal{G}$, $SEM(empty) = \emptyset$, $SEM(red(P))$ consists of all graphs G to which no rule of P can be applied, and $SEM(P, C)$ consists of all graphs G for which there is a sequence $(G_0, \ldots, G_n)$ such that $G_n = G$, for $i = 1, \ldots, n$ $(G_{i-1}, G_i) \in SEM(P)$ and $(G_0, \ldots, G_n) \in SEM_\emptyset(C)$.[8]
2. For $I, T \in \mathcal{X}$, $P \subseteq \mathcal{R}$, and $C \in \mathcal{C}$, $(I, P, C, T) \in \mathcal{X}$ where $SEM(I, P, C, T) = SEM(P, C) \cap (SEM(I) \times SEM(T))$.

One example of graph class expressions of the second type is $complete = (empty, \{nodes, edges\}, nodes^*; edges^*, red(\{edges\}))$, where *nodes* and *edges* are the rules in Fig. 2.

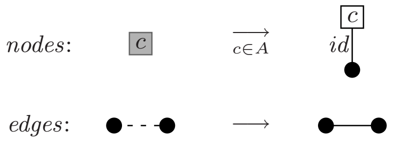


Fig. 2. The rules *nodes* and *edges*

Given some alphabet A, the expression *complete* specifies all complete graphs composed of round nodes in which every round node is additionally connected to exactly one uniquely labeled boxed node via an *id*-edge. The labels of the boxed nodes are taken from A. It is worth noting that the rule *edges* cannot produce loops because we only use injective morphisms to choose a matching of the left-hand side. In addition, we technically distinguish between round and boxed nodes by using particularly labeled loops that indicate the respective node type (*round* or *boxed*).

4 Communities of Autonomous Units

Every community is mainly composed of a set of autonomous units that act and interact in a common environment (see e.g. [HKK09] where a sequential and a parallel semantics of communities is introduced).

[8] Control conditions can be used to define sequences of graphs (instead of sequences of graph pairs) because, as stated before, rules can be regarded as rule pairs with empty private component.

4.1 Autonomous Units

Autonomous units transform a common graph and have an additional private graph where they can store private information. Since the rule set of an autonomous unit can be very large, structuring concepts should be provided to keep it manageable. Autonomous units allow to import auxiliary units and provide control conditions as well as graph class expressions. Auxiliary units differ from autonomous units in the sense that they do not contain graph class expressions. The graph class expressions of every autonomous unit are used to specify the initial private states as well as the goal. The latter consists of a private goal concerning the private state and a goal concerning the common environment that the autonomous unit wants to reach.

Definition 11 (Autonomous units).

1. A *unit* of *import depth* 0 is a system $unit = (I, U, P, C, g)$ where $I \in \mathcal{X}$ is the *initial private graph class expression*, $U = \emptyset$ is the empty set, $P \subseteq \widetilde{\mathcal{R}}$ is a set of rule pairs, $C \in \mathcal{C}(P \cup U)$ is a control condition, and $g \in \mathcal{X} \times \mathcal{X}$ is the *goal*.
2. A *unit* of *import depth* $n+1$ ($n \in \mathbb{N}$) is a system $unit = (I, U, P, C, g)$ where U is a set of units of import depth at most n, and I, P, C, and g are defined as in point 1.
3. (I, U, P, C, g) is an *auxiliary unit* if $I = all$, $g = (all, all)$, and every $u \in U$ is an auxiliary unit.
4. (I, U, P, C, g) is an *autonomous unit* if every $u \in U$ is an auxiliary unit. The set of autonomous units is denoted by AUT
5. The components of *unit* are also denoted by I_{unit}, U_{unit}, P_{unit}, C_{unit}, and g_{unit}, respectively.

Every unit can be converted into a flattened unit with import depth zero. The rule set and the control condition of the flattened unit can be constructed as follows.

Definition 12 (Flattening). For $unit = (I, U, P, C, g)$ its *flattened rule set* $Rules(unit)$ and its *flattened control condition* $flC(unit)$ is defined as follows. If $U = \emptyset$, $Rules(unit) = P$ and $flC(unit) = C$. If $U \neq \emptyset$, $Rules(unit) = P \cup \bigcup_{u \in U} Rules(u)$ and $flC(unit) = C[a]$ where $a{:}\, U \to \mathcal{C}(\widetilde{\mathcal{R}})$ is defined as $a(u) = flC(u)$.[9]

The parallel semantics of autonomous units consists of all transformation sequences that start with a pair consisting of an initial private graph and an arbitrary common environment and that are allowed by the flattened control condition. Like for control conditions, we assume the existence of a set of metarules specifying the common environment transformations that can be performed by other units. If the transformation reaches the goal, it is called successful.

[9] For a control condition c and a mapping $a{:}\, U \in \mathcal{C}$, $C[a]$ is obtained by replacing every occurrence of u with $a(u)$, for all $u \in U$.

Definition 13 (Parallel semantics). Let $aut = (I, U, P, C, (g_1, g_2))$ be an autonomous unit, let $\mathcal{MR} \subseteq \mathcal{R}_*$, and let $s = ((G_0, G'_0), \ldots, (G_n, G'_n))$ be a sequence of graph pairs. Then $s \in PAR_{\mathcal{MR}}(aut)$ if $G'_0 \in SEM(I)$ and $s \in SEM_{\mathcal{MR}}(flC(aut))$. Moreover, s is *successful* if $(G_n, G'_n) \in SEM(g_1) \times SEM(g_2)$.

Remark. In general, the transformation processes of autonomous units may also be infinite which is appropiate to describe infinite processes and in particular to investigate convergence behavior of ant-based systems. However, in this first approach we consider only the finite case, but an extension to the infinite case is straightforward (cf. [HKK09]).

A community consists of a set of autonomous units, a specification of all possible initial environments, a global control condition, and an overall goal. In the following, global control conditions are regular expressions equipped with the parallel operator $||$.

Definition 14 (Global control conditions). Let $Aut \subseteq AUT$. Then the set of *global control conditions* $\mathcal{GLC}(Aut)$ is recursively defined as follows.

1. $Aut \cup \{aut_1 || \cdots || aut_k \mid aut_i \in Aut, i = 1, \ldots, k\} \subseteq \mathcal{GLC}(Aut)$
2. For $c, c_1, c_2 \in \mathcal{GLC}(Aut)$, we have $(c_1 + c_2), (c_1 \,; c_2)$, $(c^*) \in \mathcal{GLC}(Aut)$.

Global control conditions specify sequences of states where every state consists of a common environment plus a private state for every autonomous unit in a community. Roughly speaking, the global control condition aut specifies all transformation processes of aut where the private states of all other units are not changed. The global control condition $aut_1 || \cdots || aut_k$ prescribes the parallel running of $aut_1, \ldots, aut_k$. The semantics of the remaining control conditions are defined as expected. In the following we define Aut-states and the semantics of global control conditions.

Definition 15 (Aut-states and semantics of global control conditions). For $Aut \subseteq AUT$, an *Aut-state* is a pair (G, map) where $G \in \mathcal{G}$ and $map \colon Aut \to \mathcal{G}$ is a mapping. The *semantics* of each global control condition in $\mathcal{GLC}(Aut)$ is defined as follows.

1. $SEM_{Aut}(aut)$ consists of all sequences $((G_0, map_0), \ldots, (G_n, map_n))$ of Aut-states such that $((G_0, map_0(aut)), \ldots, (G_n, map_n(aut))) \in SEM_{\emptyset}(flC(aut))$, and for each $aut' \in Aut - \{aut\}$, $map_0(aut') = \cdots = map_n(aut')$.
2. $SEM_{Aut}(aut_1 || \cdots || aut_k)$ consists of all $((G_0, map_0), \ldots, (G_n, map_n))$ such that for $i = 1, \ldots, k$,
$$((G_0, map_0(aut_i)), \ldots, (G_n, map_n(aut_i))) \in SEM_{\mathcal{MR}(aut_i)}(flC(aut_i)),$$
where $\mathcal{MR}(aut_i) = (\bigcup_{aut \in \{aut_1, \ldots, aut_k\} - \{aut_i\}} Rules(aut))_*$, and for each $aut \in Aut - \{aut_1, \ldots, aut_k\}$, $map_0(aut) = \cdots = map_n(aut)$.
3. $SEM_{Aut}(c_1 + c_2) = SEM_{Aut}(c_1) \cup SEM_{Aut}(c_2)$,
4. $SEM_{Aut}(c_1 \,; c_2) = SEM_{Aut}(c_1) \circ SEM_{Aut}(c_2)$, and
5. $SEM_{Aut}(c^*) = SEM_{Aut}(c)^*$.

The components of communities are given in the following definition.

Definition 16 (Community). A *community* is a tuple $(\mathit{Init}, \mathit{Aut}, \mathit{Cond}, \mathit{Goal})$ where $\mathit{Init}, \mathit{Goal} \in \mathcal{X}$, $\mathit{Aut} \subseteq AUT$, and $\mathit{Cond} \in \mathcal{GLC}(\mathit{Aut})$.

The parallel semantics of a community consists of all state sequences that are allowed by the global control condition and start with an initial state consisting of an initial common environment and an initial private state for each autonomous unit. The state sequences are successful if they reach the overall goal.

Definition 17 (Parallel community semantics). Let

$$\mathit{COM} = (\mathit{Init}, \mathit{Aut}, \mathit{Cond}, \mathit{Goal})$$

be a community. Then the *parallel community semantics* of COM, denoted by $PAR(\mathit{COM})$ consists of all Aut-state sequences $s = ((G_0, map_0), \ldots, (G_n, map_n))$ such that $G_0 \in \mathit{SEM}(\mathit{Init})$, $map_0(aut) \in \mathit{SEM}(I_{aut})$ (for each $aut \in \mathit{Aut}$), and $s \in \mathit{SEM}_{\mathit{Aut}}(\mathit{Cond})$. Moreover, s is *successful* if $G_n \in \mathit{SEM}(\mathit{Goal})$.

5 An ACO Community for Solving the CVRP

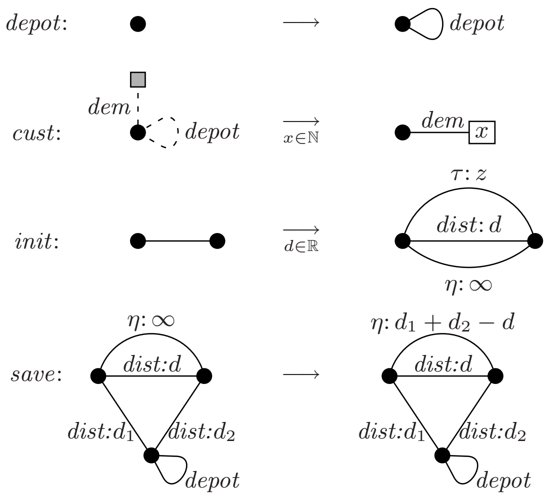
In this section we present the components of the ACO community COM_{CVRP} for modeling the Capacitated Vehicle Routing Problem (CVRP) introduced in Section 2. The initial environment specification of COM_{CVRP} specifies the construction graph of the problem; the set of autonomous units consists of the autonomous units $\mathit{Ant}_1, \ldots, \mathit{Ant}_k$ ($k \in \mathbb{N}$), and *Evap&Select*; and the global control condition *Cond* is equal to $(\mathit{Ant}_1 || \ldots || \mathit{Ant}_k || \mathit{Evap\&Select})^*$. In our first approach the overall goal is equal to *all*.

Roughly speaking, the community COM_{CVRP} works as follows. The ant units $\mathit{Ant}_1 \ldots \mathit{Ant}_k$ model the ants, which in parallel traverse the graph according to the savings heuristics introduced in Section 2 and the current pheromone trails, and search for a solution for the CVRP. When all ants have finished their search, the autonomous unit *Evap&Select* first carries out evaporation of the current pheromone trails. After that it selects the w best solutions. Now each ant which provides one of the best solutions leaves a pheromone trail on its solution path according to the quality of the solution. All the units act in parallel. To ensure the described order we use negative application conditions as well as control conditions.

5.1 The Initial Environment

The underlying structure of the construction graph of the ACO system modeling the CVRP is a complete graph with some additional information such as initial pheromone concentration, distances, etc. Therefore the construction graph for the CVRP can be defined by the graph class expression depicted in Fig. 3. It uses as initial expression the graph class expression *complete* introduced in 3.3.

Construction_graph
initial: *complete*
rules:



conds: *depot* ; $(cust + init + save)^*$
goal: $red(\{init, save, cust\})$

Fig. 3. The graph class expression *Construction_graph*

Its rule *depot* selects the depot and has to be applied exactly once. The rule *cust* adds a number representing the demand to every customer node, i.e., to every node apart from the depot. The rule *init* labels every edge e of the initial graph with a *distance* d and it inserts two edges between each two nodes of the graph, one labeled with the *heuristic value* ∞ the other with an *initial pheromone value* z. The rule *save* computes the heuristic value of every edge based on the savings heuristics. The control condition requires that the depot is selected first. The terminal graph class expression $red(\{init, save, cust\})$ guarantees that the rules *cust*, *init*, and *save* are applied as long as possible.

5.2 The Ant Units

In general, every ant builds a solution tour by traversing the common environment according to the current pheromone trails. It first selects its initial position. Afterwards, it constructs a solution tour t. Then it puts some pheromone on t if it is selected to do so. Every ant unit Ant_j uses the auxiliary units $tour_j$, and put_phero_j. The control condition is equal to $initial\_position_j$; $tour_j$; put_phero_j where $initial\_position_j$ is the rule depicted in Fig. 4. It puts the ant Ant_j to the depot and generates its memory M_j where it stores the current load of the vehicle represented by Ant_j (*load*), the capacity of the vehicle (*cap*), its current location (*sit*) and the total length of the tours (*len*). This information is represented by edges labeled with the respective labels (*load*, *cap*, *sit* and *len*), which are each attached to a node labeled with the corresponding value.

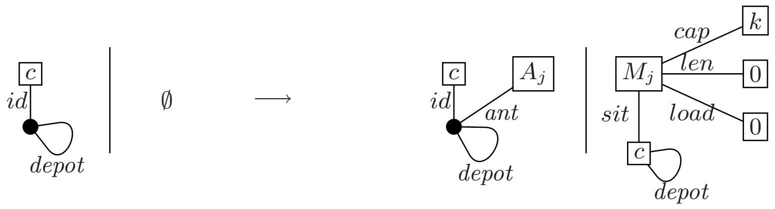


Fig. 4. The rule *initial_position*$_j$

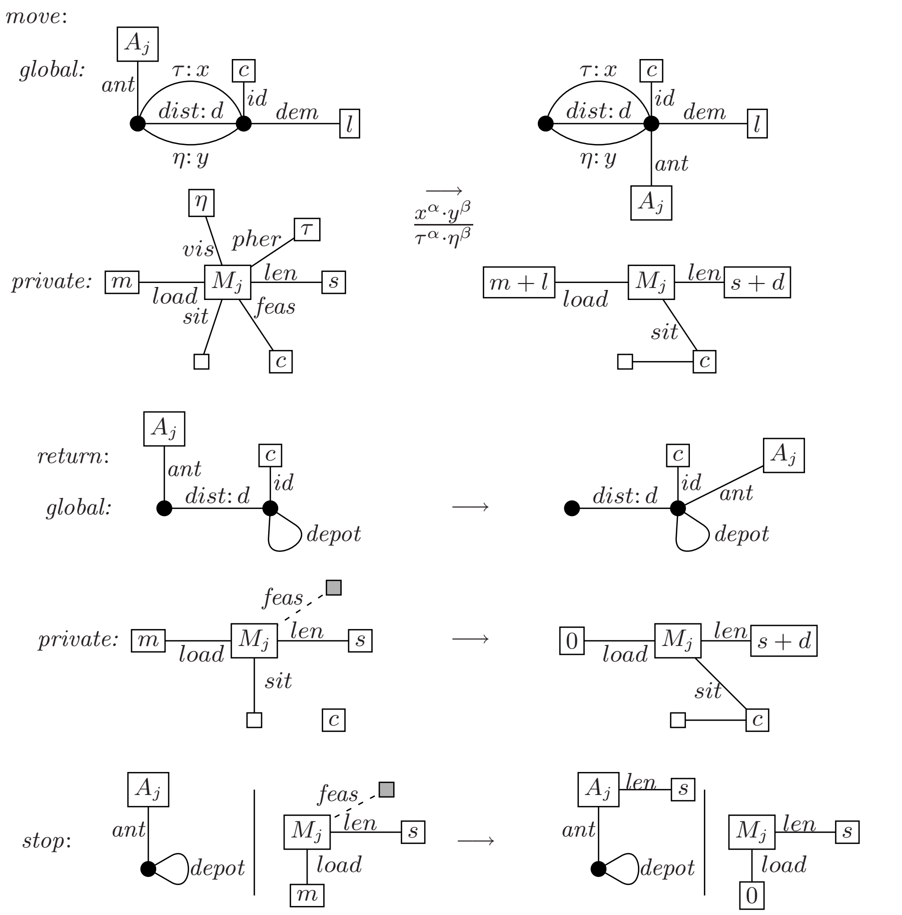
The unit *tour*$_j$ is given in Fig. 5. The global and private parts of the unit's rules are depicted one below the other. With *tour*$_j$ the ant builds a solution tour depending on probabilities for the next move to a feasible neighbour calculated from the savings heuristics and the current pheromone trails. It contains the auxiliary units *feasible_neighbours*$_j$ and *prob*$_j$, and the rules *move*, *return* and *stop*. The control condition requires to apply the unit *feasible_neighbours*$_j$ first. This unit is given is given in Fig. 6. It computes the feasible neighbours for an ant unit *Ant*$_j$ and stores them in the memory of the ant. Feasible neighbours are customer-nodes that are not yet visited and whose demand still fits into the vehicle. Every application of the only rule *feas* adds one feasible neighbour to the memory. Moreover, it uses the auxiliary unit *delete_nonfeasible* that removes all neighbours from the memory that are connected via a *feas*-edge to M_j and whose demand exceeds the remaining capacity of the vehicle.[10] This is necessary because after adding a feasible customer to a tour, the former feasible neighbours may not fit into the vehicle anymore. For reasons of space limitations a drawing of *delete_nonfeasible* is omitted.

The unit *prob*$_j$ is given in Fig. 7. It provides some of the values that are needed by the unit *tour*$_j$ for computing the probability that a feasible neighbour is chosen for a next move. This is done by summing up the pheromone and the heuristic values of all feasible neighbours. The rule *begin* initializes these values with 0. The rule *sum* must be applied as long as possible. For not counting a feasible neighbour several times *sum* changes each label *feas* into *ok*. At the end the unit *relabel_all_private*$_j$ *(ok,feas)* is applied which undoes this relabeling, i.e., it changes all *ok*-edges into *feas*-edges. It is very simple and hence not depicted.

With the rule *move* the ant moves to a feasible neighbour with the probability depicted under the arrow of the rule *move* in Fig. 5. Moreover, in the memory the current load of the vehicle, the path followed so far, and the total length of the tour are updated. With the rule *return* the ant returns to the depot if no feasible neighbour is left and resets its current load to 0. Afterwards it starts to construct a new subtour. Finally, when all nodes are visited, the rule *stop*

[10] We assume that the demand of each customer fits into one vehicle.

tour$_j$
uses: *feasible_neighbours*$_j$, *prob*$_j$
rules:



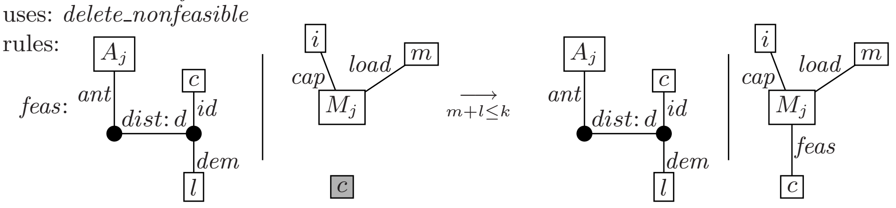
conds: (*feasible_neighbours*$_j$; (*prob*$_j$; *move* + *return*))* ; *feasible_neighbours*$_j$; *stop*

Fig. 5. The auxiliary unit *tour*$_j$

is applied to reset the current load in the memory of the ant to 0. The rule *stop* also it adds the information about the length of the found solution to the common environment by inserting an edge labeled with *len* from the ant-node A_j to a new node labeled with the length of the solution.

The unit *put_phero*$_j$ is depicted in Fig. 8. It works a little different for ants, who should leave a pheromone trail and those who should not. Both kinds of ants apply different rules, but the structure of rule applications is the same. In both

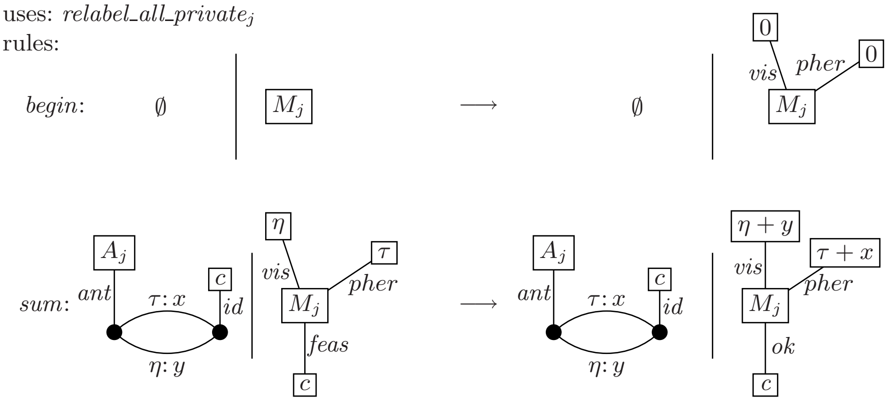
feasible_neighbours$_j$
uses: *delete_nonfeasible*
rules:



conds: *delete_nonfeasible* ; *feas*!

Fig. 6. The auxiliary unit *feasible_neighbours*$_j$

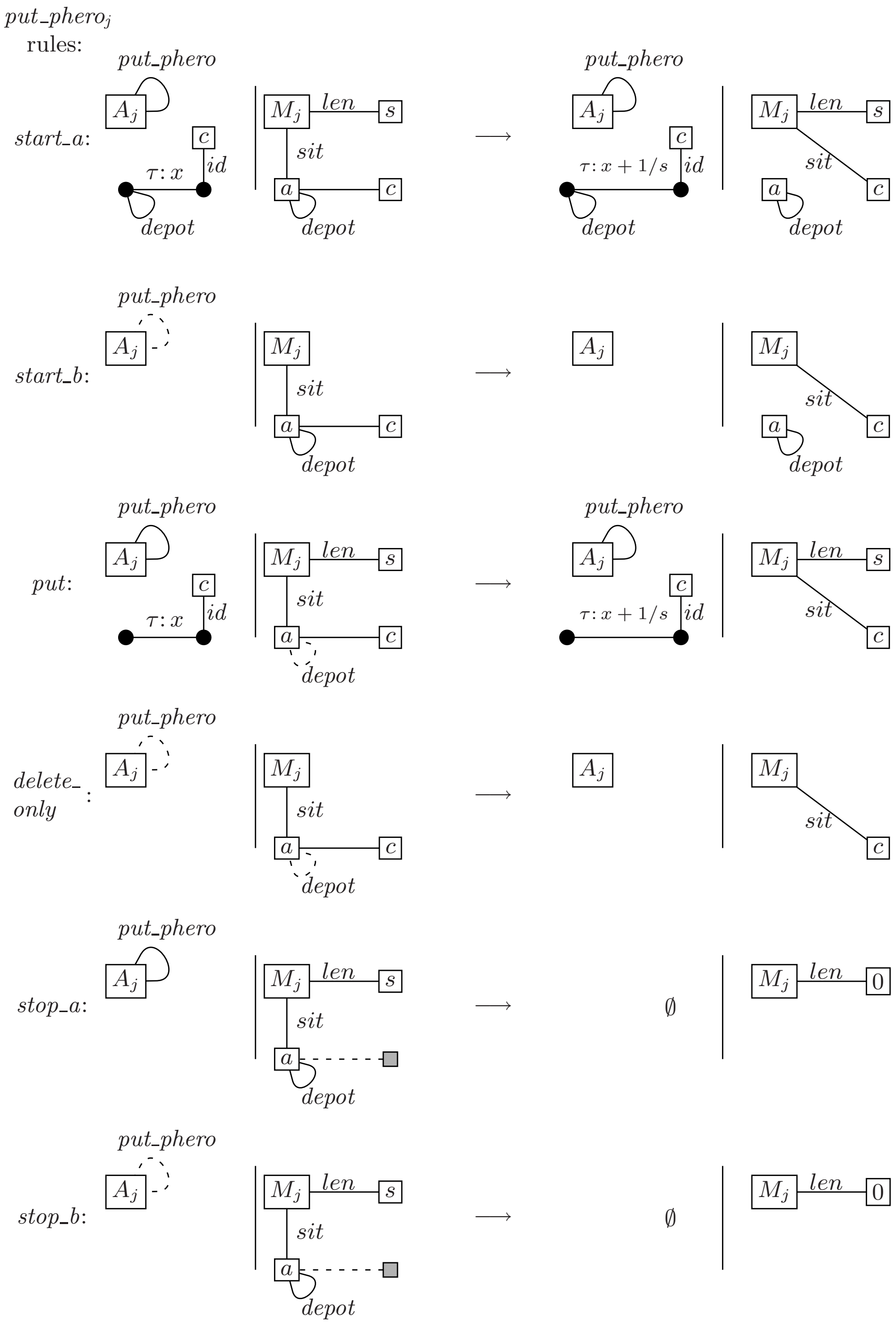
prob$_j$
uses: *relabel_all_private*$_j$
rules:



conds: *begin* ; *sum*! ; *relabel_all_private*$_j$ *(ok,feas)*

Fig. 7. The auxiliary unit *prob*$_j$

cases the ant traverses the solution path stored in its memory and meanwhile deletes it. (Because the path stored in the memory is shaped like a blossom with the depot in the middle, first the "petals" (subtours) are deleted and finally the depot.) This behaviour is represented by the rules *start_a* (resp. *start_b*) and *put* (resp. *delete_only*) and the subexpression of the control condition ((*start_a* + *start_b*) ; (*put*! + *delete_only*!))*. One application of a *start*-rule followed by applications of the rule *put* (resp. *delete_only*) as long as possible traverses one subtour of the found tour beginning and ending at the depot. The rules delete the traversed path from the memory (leaving the depot); *put* additionally leaves a pheromone trail in the common environment with the value $1/s$, where s is the length of the solution tour. Afterwards the remaining subtours are traversed until no further subtour is left in the memory. Then the respective *stop*-rule can be applied, which deletes the ant A_j from the common environment, the depot from the memory and resets the length of the traversed path to 0.



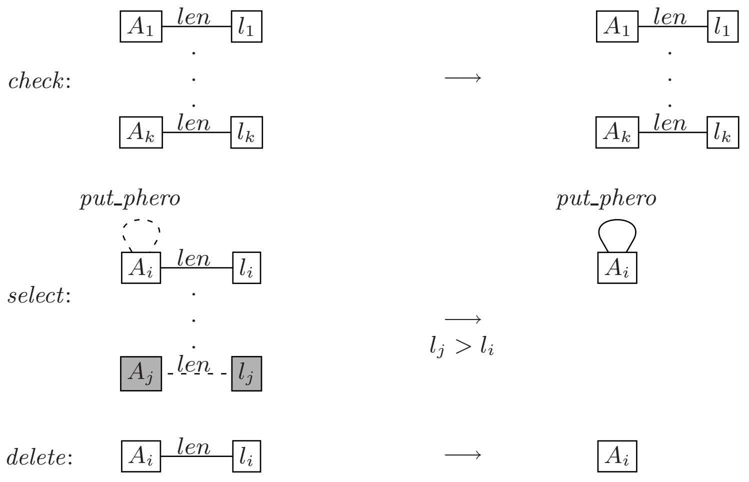
conds: $((start\_a + start\_b) \; ; \; (put! + delete\_only!))^{*} \; ; \; (stop\_a + stop\_b)$

Fig. 8. The auxiliary unit *put_phero$_j$*

5.3 The Unit *Evap&Select*

Evap&Select is given in Fig. 9. It is responsible for the evaporation of old pheromone trails, for the selection of the w best solutions provided by the ants, and for marking these w ants with a *put_phero*-label.

Evap&Select
uses: *relabel_all_global*
rules:



conds: *check* ; *relabel_all_global(τ: z,τ: $(1-\rho)*z$)* ; *select*^{w} ; *delete*!

Fig. 9. The autonomous unit *Evap&Select*

With the rule *check*, which is applied only once, the unit checks whether all ants have finished their search. This is the case if all ants have written the length of the found solution into the common environment. With the help of the unit *relabel_all_global* evaporation takes place by multiplying the pheromone value of every pheromone edge in the common environment with $(1-\rho)$, where $\rho \in (0,1]$ is a pheromone decay parameter. After that, the rule *select* is applied w times (in the control condition this is abbreviated by *select*^{w}). The rule *select* finds the ant with the best solution, marks it with a label *put_phero*, and deletes the information about the length of the ant's solution from the common environment. Each further application of *select* finds the next best solution. When the w best solutions are found, the rule *delete* is applied as long as possible to delete the remaining nodes and edges displaying the information about the lengths of the ants' solutions. This rank-based approach could be extended by the *elitist strategy* (see e.g. [DS04]). In this strategy the best solution so far is memorized and when pheromone update takes place, this tour gets additional pheromone.

(In our modeling of the CVRP, we do not consider this strategy because of space limitations.)

Remark. The presented modelization can be used to prove correctness properties a few of which are informally described here.

- In every transformation sequence of $tour_j$ a solution is constructed, i.e., a set of cycles of the construction graph is traversed by Ant_j and stored in its memory such that the depot belongs to every cycle, and every customer occurs exactly once in exactly one cycle.
- The unit put_phero_j deletes the constructed solution from the memory of Ant_j and increases the pheromone value of each edge in the solution by $\frac{1}{s}$ where s is the length of the solution.
- The unit feasible neighbours stores all nodes in the memory of Ant_j that are not visited, yet.
- Each execution of $(Ant_1|| \dots ||Ant_k||Evap\&Select)$ models an iteration of the corresponding ACO-Algorithm, i.e., (1) solution construction, (2) pheromone update, and (3) evaporation.

6 Conclusion

In this paper, we have modeled an ACO algorithm for the Capacitated Vehicle Routing Problem as a community of autonomous units. The autonomous behavior of every ant has been modeled as an autonomous unit, and global features of ACO algorithms such as the construction graph or the order in which solution construction, pheromone update, and evaporation take place have been modeled with global components of communities such as the initial environment specification or the global control condition. Since all ACO algorithms basically work according to the same underlying algorithm, we believe that they all can be modeled as communities of autonomous units in a natural way.

For solving ACO algorithms in a proper way, we have extended the parallel working autonomous units of [HKK09] by auxiliary units that allow to encapsulate auxiliary tasks in separate units and to manage large rule sets. We also have added a separate state for every autonomous unit in order to represent memories of ants. Furthermore, we have defined the syntax and a proactive semantics of a concrete class of control conditions that is adequate for units running in parallel. This class consists of regular expressions extended by a parallelism operator and an operator that prescribes to apply a rule as long as possible. We have given a construction that flattens the (hierarchical) import structure and the control conditions of autonomous units so that the parallel semantics of [HKK09] could be used for the extended units.

The modeling of ACO systems as communities of autonomous units has the following advantages. (1) The specification of ants as autonomous units provides the ants with a well-defined operational semantics. (2) The graph transformation rules of autonomous units allow for a visual specification of ant behavior instead of string-based pseudo code as it is often used in the literature. (3) The existing graph transformation systems (cf. e.g. [ERT99,GK08]) facilitate the visual simulation of ant colonies in a straightforward way (see also [Höl08]). (4) The formal semantics of communities of autonomous units constitutes a basis for proving correctness results by induction on the length of the transformation sequences or for examining other characteristics (such as termination) by making use of the wide theory of rule-based graph transformation (see [Roz97]). (5) Implementing ACO algorithms with graph transformational systems is useful for verification purposes, i.e., to check whether the algorithms behave properly for specific cases.

In the future, this and further case studies should be implemented with one of the existing graph transformation systems so that (1) the emerging behavior of ant colonies can be visually simulated, and (2) ACO algorithms can be verified. For the implementation purpose we plan to use GrGen [GK08] because it is one of the fastest and most flexible graph transformation systems. Further case studies could take into account more advanced elitist strategies as well as dynamic aspects (see e.g. [ES02,DS04,MGRV05,RDH04,RMLG07]). Another interesting task is to investigate how communities of autonomous units can serve as a modeling framework for swarm intelligence in general.

References

[CEH+97] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg [Roz97], pages 163–245.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization.* MIT-Press, 2004.

[ERT99] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG-approach: Language and environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*, pages 551–603. World Scientific, Singapore, 1999.

[ES02] Casper Joost Eyckelhof and Marko Snoek. Ant systems for a dynamic TSP - ants caught in a traffic jam. In M. Dorigo, G. Caro Di, and M. Sampels, editors, *Ant Algorithms - Third International Workshop, ANTS 2002*, volume 2462 of *Lecture notes in Computer Science*, pages 88–98, 2002.

[GK08] Rubino Geiß and Moritz Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, volume 5088 of *Lecture Notes in Computer Science*, pages 568–569, 2008.

[HKK09] Karsten Hölscher, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous units to model interacting sequential and parallel processes. *Fundamenta Informaticae*, 92(3):233–257, 2009.

[Höl08] Karsten Hölscher. *Autonomous Units as a Rule-based Concept for the Modeling of Autonomous and Cooperating Processes.* Logos Verlag, 2008. PhD thesis.

[KK07] Hans-Jörg Kreowski and Sabine Kuske. Autonomous units and their semantics - the parallel case. In J.L. Fiadeiro and P.Y. Schobbens, editors, *Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006*, volume 4408 of *Lecture Notes in Computer Science*, pages 56–73, 2007.

[KK08] Hans-Jörg Kreowski and Sabine Kuske. Communities of autonomous units for pickup and delivery vehicle routing. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, volume 5088 of *Lecture Notes in Computer Science*, pages 281–296, 2008.

[MGRV05] Roberto Montemanni, Luca Maria Gambardella, Andrea Emilio Rizzoli, and Alberto V.Donati. Ant colony system for a dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 10(4):327–343, 2005.

[RDH04] Marc Reimann, Karl Doerner, and Richard F. Hartl. D-ants: Savings based ants divide and conquer the vehicle routing problem. *Computers & OR*, 31(4):563–591, 2004.

[RMLG07] Andrea Emilio Rizzoli, Roberto Montemanni, Enzo Lucibello, and Luca Maria Gambardella. Ant colony optimization for real-world vehicle routing problems. *Swarm Intelligence*, 1(2):135–151, 2007.

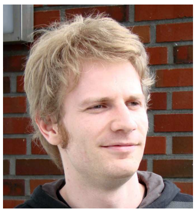
[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific, Singapore, 1997.

Dr. Sabine Kuske
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
kuske@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~kuske

Sabine Kuske has been a member of Hans-Jörg's team since November 1991. He supervised her diploma and doctoral theses. Their common research interests concern all aspects of graph transformation; in particular, they have been working together on autonomous units.

Melanie Luderer
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
melu@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~melu

Melanie Luderer is a doctoral student of the International Graduate School for Dynamics in Logistics since 2006, and Hans-Jörg Kreowski is her supervisor. He was also the supervisor for her diploma thesis.



Hauke Tönnies
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
hatoe@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/˜hatoe

Hauke Tönnies is a doctoral student supervised by Hans-Jörg Kreowski since 2008. He is supported by the Collaborative Research Centre 637: Autonomous Cooperating Logistic Processes.