Autonomous Units and Their Semantics – The Concurrent Case[*]

Hans-Jörg Kreowski and Sabine Kuske

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

Abstract. Communities of autonomous units are rule-based and graph-transformational devices to model processes that act and interact, move and communicate, cooperate and compete in a common environment. The autonomous units are independent of each other, and the environment may be large and structured in such a way that a global synchronization of process activities is not reasonable or not feasible. To reflect this assumption properly, a concurrent-process semantics of autonomous units is introduced and studied in this paper employing the idea of true concurrency. In particular, causal dependency between actions of autonomous units is compared with shift equivalence known from graph transformation, and concurrent processes in the present approach are related to canonical derivations.

1 Introduction

In this paper, we introduce and investigate the concurrent semantics of autonomous units. Communities of autonomous units are proposed in [7] as rule-based and graph-transformational devices to model interactive processes that run independently of each other in a common environment. An autonomous unit has a goal that it tries to reach, a set of rules the applications of which provide its actions, and a control condition which regulates the choice of actions to be performed actually. Each autonomous unit decides about its activities on its own right depending on the state of the environment and the possibility of rule applications, but without direct influence of other ongoing processes.

In [9], the sequential as well as the parallel semantics of autonomous units is studied. In the sequential case, a single unit can act at a time while all other units must wait. This yields sequences of rule applications interleaving the activities of the various units. Typical examples of this kind are board games with several players who can perform their moves in turn. In the parallel case, the process steps are given by the application of parallel rules that are composed of the rules of the active units. In this way, units can act simultaneously providing a kind

[*] Research 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).

of parallelism which is known from Petri nets, cellular automata, multi-agent systems, and graph transformation.

The sequential and the parallel semantics of communities of autonomous units are based on sequential and parallel derivations respectively. Both are composed of derivation steps. In other words, the semantics assumes implicitly the existence of a global clock to cut the run of the whole system into steps. But this is not always a realistic assumption, because the environment may be very large and – more important – the idea of autonomy conflicts with the regulation by a global clock. For example, trucks in a large transport network upload, move, and deliver asynchronously, and do not operate in simultaneous steps and even less in interleaved sequential steps.

The concurrent semantics avoids the assumption of a global clock. The actions of units are no longer totally ordered or simultaneous, but only partially ordered. The partial order reflects causal dependencies meaning that one action takes place before another action if the latter needs something that the former provides. The causal dependency relation of the concurrent semantics of autonomous units is compared with shift equivalence known from graph transformation, and concurrent processes in the present approach are related to canonical derivations (see also [11,3,2]).

The paper is organized as follows. Section 2 contains some preliminaries concerning multisets and graphs. In Section 3, concurrent graph transformation approaches are introduced which provide the basic ingredients of autonomous units. In Section 4, communites of autonomous units are presented and a concurrent semantics is defined for them. Section 5 is dedicated to canonical derivations which constitute a kind of representantives for the concurrent runs in a community. The introduced concepts are illustrated with a running example solving a generalization of the well-known Hamiltonian path problem. The last section contains the conclusion.

2 Preliminaries

In this section, we recall some basic notions and notations concerning multisets and graphs as far as they are needed in this paper.

Multisets. Given some basic domain D, the set of all multisets D_* over D with finite carriers consists of all mappings $m: D \to \mathbb{N}$ such that the carrier $car(m) = \{d \in D \mid m(d) \neq 0\}$ is finite. For $d \in D$, $m(d)$ is called the multiplicity of d in m. The union or sum of multisets can be defined by adding corresponding multiplicities, i.e., $m + m'(d) = m(d) + m'(d)$ for all $m, m' \in D_*$ and $d \in D$. D_* with this sum is the free commutative monoid over D where the multiset with empty carrier is the null element, i.e. $\mathbf{0}: D \to \mathbb{N}$ with $\mathbf{0}(d) = 0$ for all $d \in D$. Note that the elements of D correspond one-to-one to singleton multisets, i.e. for $d \in D, \hat{d}: D \to \mathbb{N}$ with $\hat{d}(d) = 1$ and $\hat{d}(d') = 0$ for $d' \neq d$. These singleton multisets are the generators of the free commutative monoid. This means in particular that, for every $m \in D_*$, there are $d_1, \ldots, d_k \in D$ with $m = \sum_{i=1}^{k} \widehat{d_i}$.

Graphs. A (directed edge-labeled) graph is a system $G = (V, E, s, t, l)$ where V is a set of nodes, E is a set of edges, $s, t: E \to V$ assign to every edge its source $s(e)$ and its target $t(e)$, and the mapping l assigns a label to every edge in E. The components of G are also denoted by V_G, E_G, etc. As usual, a graph M is a subgraph of G, denoted by $M \subseteq G$ if $V_M \subseteq V_G$, $E_M \subseteq E_G$, and s_M, t_M, and l_M are the restrictions of s_G, t_G, and l_G to E_M. A graph morphism $g: L \to G$ from a graph L to a graph G consists of two mappings $g_V: V_L \to V_G$, $g_E: E_L \to E_G$ such that sources, targets and labels are preserved, i.e. for all $e \in E_L$, $g_V(s_L(e)) = s_G(g_E(e))$, $g_V(t_L(e)) = t_G(g_E(e))$, and $l_L(e) = l_G(g_E(e))$. In the following we omit the subscript V or E of g if it can be derived from the context. In order to represent also graphs that contain labeled as well as unlabeled edges, we assume the existence of a special symbol $\diamond$. Every edge labeled with $\diamond$ is then regarded as an *unlabeled edge*. All other edges are *labeled edges*. An edge is called a *loop* if its source and target coincide. In graphical representations we omit the loops, i.e., their labels are placed next to their sources. If the labeled loops of a node is the set $\{e_1, \ldots, e_k\}$ where for $i = 1, \ldots, k$ the label of e_i is x_i ($x_i \neq \diamond$), the node will be called a $\{x_1, \ldots, x_k\}$-node. In the case where $k = 1$, the node is called an x_1-node. A node without any labeled loop is called a λ-node.

3 Concurrent Graph Transformation Approaches

Graph transformation (see, e.g., [18,6,1]) constitutes a formal specification framework that supports the modeling of the rule-based transformation of graph-like, diagrammatic, and visual structures. The ingredients of graph transformation are provided by so-called graph transformation approaches. In this section, we recall the notion of a graph transformation approach as introduced in [12], but modified with respect to the purposes of this paper.

Two basic components of every graph transformation approach are a class of graphs and a class of rules that can be applied to these graphs. In many cases, rule application is highly nondeterministic – a property that is not always desirable. Hence, graph transformation approaches can also provide a class of control conditions so that the degree of nondeterminism of rule application can be reduced. Moreover, graph class expressions can be used in order to specify sets of initial and terminal graphs of graph transformations. In the following, transformations from initial to terminal graphs via rule applications according to control conditions are called (graph transformation) processes.

The basic idea of parallelism in a rule-based framework is the application of many rules simultaneously and also the multiple application of a single rule. To achieve these possibilities, we assume that multisets of rules can be applied to graphs rather than single rules. If $\mathcal{R}$ is a set of rules, $r \in \mathcal{R}_*$ comprises a selection of rules each with some multiplicity. Therefore, an application of r to a graph yields a graph which models the parallel and multiple application of several rules.

If there is no global clock and no synchronization mechanism that cuts the actions (i.e., the rule applications) of processes into steps, then the process actions are not totally ordered, but only partially. An action occurs necessarily before another action if the former one generates something that the latter one needs. Moreover, an action that removes something that is needed by another action prohibits the latter one to occur if the former one is performed first. Both situations describe causal dependencies. In all other cases, it may be impossible to establish the order of time of two actions: They may occur one after the other in any order or even in parallel. This is the idea of true concurrency which we employ to define concurrent graph transformation.

Definition 1 (Concurrent graph transformation approach). A concurrent graph transformation approach is a system $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \mathcal{X}, \mathcal{C})$ the components of which are the following.

– $\mathcal{G}$ is a class of *graphs*.
– $\mathcal{R}$ is a class of *graph transformation rules* such that every $r \in \mathcal{R}_*$ specifies a binary relation on graphs $\mathit{SEM}(r) \subseteq \mathcal{G} \times \mathcal{G}$, which is subject to the following *true-concurrency condition*: $(G, G'') \in \mathit{SEM}(r_1 + r_2)$ for $r_1, r_2 \in \mathcal{R}_*$ implies $(G, G') \in \mathit{SEM}(r_1)$ and $(G', G'') \in \mathit{SEM}(r_2)$ for some $G' \in \mathcal{G}$. Moreover, we assume that $(G, H) \in \mathit{SEM}(\mathbf{0})$ implies $G = H$ for the null element $\mathbf{0} \in \mathcal{R}_*$.
– $\mathcal{X}$ is a class of *graph class expressions* such that each $x \in \mathcal{X}$ specifies a set of graphs $\mathit{SEM}(x) \subseteq \mathcal{G}$.
– $\mathcal{C}$ is a class of *control conditions* such that each $c \in \mathcal{C}$ specifies a set of graph pairs $\mathit{SEM}_{\mathcal{AR},P}(c) \subseteq \mathcal{G} \times \mathcal{G}$ for every set $\mathcal{AR} \subseteq \mathcal{R}_*$ and every set $P \subseteq \mathcal{R}$.

Remarks.

1. The multisets of rules in $\mathcal{R}_*$ are called *parallel rules*. A pair of graphs $(G, G') \in \mathit{SEM}(r)$ for some $r \in R_*$ is an application of the parallel rule r to G with the result G'. It may be also called a direct parallel derivation or a parallel derivation step denoted by $G \underset{r}{\Longrightarrow} G'$. Accordingly, a sequence of parallel derivation steps $G = G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_k}{\Longrightarrow} G_k = G'$ for $k \in \mathbb{N}$ defines a parallel derivation (of length k) from G to G', which may be denoted by $G \overset{*}{\underset{P}{\Longrightarrow}} G'$ if $r_1, \cdots, r_k \in P_*$ for $P \subseteq \mathcal{R}$.
2. The control conditions are meant to restrict the nondeterminism of rule applications so that some reference is needed to the rules in consideration. Hence, the semantics of control conditions has two rules parameters which are used in the definition of the semantics of autonomous units in the next section. On one hand, the semantics of a control condition c of an autonomous unit may depend on the rule set P of the unit itself. On the other hand, it depends on a set $\mathcal{AR}$ of *active rules* typically being the set of parallel rules that can be composed of the rules of all units in a community. Hence, the set $\mathcal{AR}$ specifies all parallel derivations that can be constructed with the rules of a community, and the rule set P indicates which of these rules belong to the unit with the control condition c.

3. Due to the true-concurrency condition, each direct parallel derivation $d = (G \underset{r_1+r_2}{\Longrightarrow} G'')$ gives rise to a parallel derivation $G \underset{r_1}{\Longrightarrow} G' \underset{r_2}{\Longrightarrow} G''$ which is called a *sequentialization* of d. The two parallel derivation steps are called *independent* (of each other). Note that there is a second sequentialization of the form $G \underset{r_2}{\Longrightarrow} \hat{G} \underset{r_1}{\Longrightarrow} G''$ because of the commutativity in R_*. A parallel derivation step and its sequentialization can be considered as *equivalent* w.r.t. true concurrency. If this relation is closed under sequential composition of parallel derivations as well as reflexivity, symmetry, and transitivity, one gets the *true-concurrency equivalence* on parallel derivations, which is denoted by $\equiv$. This is made explicit in the following definition.

Definition 2 (True-concurrency equivalence). Let $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \mathcal{X}, \mathcal{C})$ be a concurrent graph transformation approach, and let $DER(\mathcal{A})$ be the set of all parallel derivations over $\mathcal{A}$. Then the *true-concurrency-equivalence* is recursively defined on $DER(\mathcal{A})$ as follows:

1. Let $d = (G \underset{r_1+r_2}{\Longrightarrow} G'')$ be a direct parallel derivation and let
$$d' = (G \underset{r_1}{\Longrightarrow} G' \underset{r_2}{\Longrightarrow} G'')$$
be its sequentialization. Then $d \equiv d'$.
2. Let $d = (G \overset{*}{\underset{P}{\Longrightarrow}} \overline{G})$, $d' = (G \overset{*}{\underset{P}{\Longrightarrow}} \overline{G})$, $c = (F \overset{*}{\underset{P}{\Longrightarrow}} G)$, and $e = (\overline{G} \overset{*}{\underset{P}{\Longrightarrow}} H)$ be parallel derivations for some $P \subseteq \mathcal{R}$. If $d \equiv d'$, then $c \circ d \equiv c \circ d'$ and $d \circ e \equiv d' \circ e$.[1]
3. $d \equiv d$ for all $d \in DER(\mathcal{A})$.
4. If $d \equiv d'$, then $d' \equiv d$ for all $d, d' \in DER(\mathcal{A})$.
5. If $d \equiv d'$ and $d' \equiv d''$, then $d \equiv d''$ for all $d, d', d'' \in DER(\mathcal{A})$.

For technical simplicity we assume in the following that $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \mathcal{X}, \mathcal{C})$ is an arbitrary but fixed concurrent graph transformation approach.

Examples

In the following we present some instances of the components of concurrent graph transformation approaches which are partly used in the examples of the following sections. Further examples of graph transformation approaches can be found in, e.g., [18].

Graphs. A well-known instance for the class $\mathcal{G}$ is the class of all directed edge-labeled graphs as defined in Section 2. Other classes of graphs are trees, undirected graphs, hypergraphs, etc.

Rules. As a concrete example of rules we consider the so-called DPO rules each of which consists of a triple $r = (L, K, R)$ of graphs such that $L \supseteq K \subseteq R$ (cf. [3]). The application of a rule to a graph G yields a graph G', if one proceeds according to the following steps:

[1] Here, $\circ$ denotes the sequential composition of derivations.

1. Choose a graph morphism $g: L \to G$ so that for all items x, y (nodes or edges) of L with $x \neq y$, $g(x) = g(y)$ implies that x and y are in K.
2. Delete all items of $g(L) - g(K)$ provided that this does not produce dangling edges. (In the case of dangling edges the morphism g cannot be used.) The resulting graph is denoted by D.
3. Add R to the graph D.
4. Glue D and R by identifying the nodes and edges of K in R with their images under g.

The conditions of (1) and (2) concerning g are called gluing condition.

Graph transformation rules can be depicted in several forms. In the following they are shown by drawing only its left-hand side L and its right-hand side R together with an arrow pointing from L to R, i.e. $L \to R$. The nodes of K are distinguished by different shapes and fill-styles occurring in L and R as well.

Figure 1 shows an example of a rule. To interpret the drawing properly, one should remember that loops are not drawn, but the labels of the loops are placed next to the nodes which the loops are incident with. In particular, a v-node is a node with a v-labeled loop and a $\{b, e\}$-node a node with two loops labeled with b and e, respectively. The left-hand side of the rule in Figure 1 consists of a v-node, the intermediate graph K is equal to the left-hand side, and the right-hand side consists of the same v-node, a new $\{b, e\}$-node, and a new edge labeled with v. The edge points from the v-node to the $\{b, e\}$-node. Hence, the rule can be applied to a graph with a v-node with the effect that a new $\{b, e\}$-node and a v-labeled edge from the v-node to the $\{b, e\}$- node are generated.

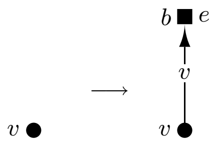

Fig. 1. A rule

Given two rules $r_i = (L_i, K_i, R_i)$ $(i = 1, 2)$, their parallel composition yields the rule $r_1 + r_2 = (L_1 + L_2, K_1 + K_2, R_1 + R_2)$ where $+$ denotes the disjoint union of graphs. In the same way, one can construct a parallel rule from any multiset $r \in \mathcal{R}_*$. For every pair $(G, G'') \in SEM(r_1 + r_2)$ there exists a graph G' such that (G, G') is in $SEM(r_1)$ and (G', G'') is in $SEM(r_2)$. This means that the applications of DPO rules are truly concurrent. (see, e.g., [3] for more details).

Graph class expressions. Every subset $M \subseteq \mathcal{G}$ is a graph class expression that specifies itself, i.e. $SEM(M) = M$. A single graph G can also serve as a graph class expression specifying all graphs G' with $G \subseteq G'$. This type of graph class expressions is called *subgraph condition.* Moreover, every set $\mathcal{L}$ of labels specifies the class of all graphs in $\mathcal{G}$ the labels of which are elements of $\mathcal{L}$. Every set $P \subseteq \mathcal{R}_*$ of (parallel) graph transformation rules can also be used as a graph class expression specifying the set of all graphs that are reduced w.r.t. P where a graph is said to be reduced w.r.t. P if no rules of P can be applied to the graph. The least restrictive graph class expression is the term *all* specifying the class $\mathcal{G}$.

Control conditions. The least restrictive control condition is the term *free* that allows all pairs of graphs, i.e. $SEM_{\mathcal{AR},P}(free) = \mathcal{G} \times \mathcal{G}$ for all $\mathcal{AR} \subseteq \mathcal{R}_*$ and all $P \subseteq \mathcal{R}$. This is the only control condition used in our running example. A much more restrictive control condition is a single rule $r \in P$. Its semantics $SEM_{\mathcal{AR},P}(r)$ consists of the initial and final graphs of all parallel derivations $G_0 \underset{r_1}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ $(n > 0)$ where $r_1, \ldots, r_n \in \mathcal{AR}$, exactly one r_i contains at least one copy of r, and no other rule of P is applied.[2] More formally, this is expressed as follows:

– There is an $i \in \{1, \ldots, n\}$ such that $r_i(r) > 0$.
– For $j = 1, \ldots, n$, $r_j(r') = 0$, for all $r' \in P \setminus \{r\}$.
– For all $j \in \{1, \ldots, n\}$ with $j \neq i$, $r_j(r) = 0$.

A more general control condition is a set $M \subseteq P$ where $SEM_{\mathcal{AR},P}(M)$ specifies the initial and terminal graphs of all parallel derivations $G_0 \underset{r_1}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ such that for $i = 1, \ldots, n$, $r_i \in \mathcal{AR}$, and $r_i(r) = 0$ if $r \in P \setminus M$. This means that in the applied parallel rules no copy of a rule in $P \setminus M$ may occur.

Since the semantics of control conditions are binary relations, they can be sequentially composed. For control conditions c and c', their sequential composition is denoted by $c\,;c'$ with $SEM_{\mathcal{AR},P}(c\,;c') = SEM_{\mathcal{AR},P}(c) \circ SEM_{\mathcal{AR},P}(c')$. Other useful control conditions are regular expressions, *as long as possible*, as well as priorities (cf. [14]).

4 Communities of Autonomous Units

Autonomous units interact in a common environment which is modeled as a graph. As a basic modeling device, an autonomous unit consists of a set of graph transformation rules, a control condition, and a goal. The graph transformation rules contained in an autonomous unit *aut* specify all transformations the unit *aut* can perform. Such a transformation comprises for example a movement of the autonomous unit within the current environment, the exchange of information with other units via the environment, or local changes of the environment. The control condition regulates the application process. For example, it may require that a sequence of rules be applied as long as possible. The goal of a unit is a graph class expression determining how the transformed graphs should look like eventually.

[2] We assume that identical rules of different autonomous units can be distinguished in $\mathcal{AR}$. This can be achieved by considering named rules. For technical simplicity, this is not further regarded in this paper.

Definition 3 (Autonomous unit). An *autonomous unit* is a system $aut = (g, P, c)$ where $g \in \mathcal{X}$ is the *goal*, $P \subseteq \mathcal{R}$ is a set of graph transformation rules, and $c \in \mathcal{C}$ is a control condition. The components of *aut* are also denoted by g_{aut}, P_{aut}, and c_{aut}, respectively.

An autonomous unit modifies an underlying environment while striving for its goal. In the setting of a concurrent graph transformation approach, its semantics consists of a set of equivalence classes of parallel derivations w.r.t. the true-concurrency equivalence. This concerns parallel derivations which comprise the parallel application of local rules of the considered unit as well as of rules performed by other autonomous units that are working in the same environment. In a concurrent setting, environment changes performed by other units must be possible while a single autonomous unit applies its rules. To cover this in the definition of the semantics, we assume a variable set of active rules that descibes all possibilities of coexisting units. Moreover, autonomous units regulate their transformation processes by choosing only those rules that are allowed by their control condition. A transformation process is called successful if its last environment satisfies the goal of the unit.

Definition 4 (Parallel and concurrent semantics).

1. Let $aut = (g, P, c)$ be an autonomous unit, let $\mathcal{AR} \subseteq \mathcal{R}_*$ be a set of parallel rules, called *active rules*, and let $d = (G \overset{*}{\underset{\mathcal{AR}}{\Longrightarrow}} G') \in DER(\mathcal{A})$ be a parallel derivation over $\mathcal{A}$. Then d is a *parallel run* of *aut* if $(G, G') \in SEM_{\mathcal{AR},P}(c)$.
2. The set of parallel runs of *aut* is denoted by $PAR_{\mathcal{AR}}(aut)$.
3. The derivation d is called a *successful parallel run* if $G' \in SEM(g)$.
4. Let $d \in PAR_{\mathcal{AR}}(aut)$. Then $[d]$ is a *concurrent run* of *aut* where $[d]$ denotes the equivalence class of d w.r.t. the true-concurrency equivalence $\equiv$, i.e., $[d] = \{d' \mid d \equiv d'\}$.
5. The set of concurrent runs of *aut* is denoted by $CONCUR_{\mathcal{AR}}(aut)$.
6. A concurrent run $[d]$ is *successful* if it contains a successful parallel run.

Remarks.

1. A parallel run of an autonomous unit depends on its rules and its control condition as the pair of the start graph and the result graph must be accepted by the control condition semantics with respect to the rules of the unit. Moreover, it depends on the set $\mathcal{AR}$ of active rules that reflects the context of the considered unit. The definition of the parallel and concurrent semantics does not fix the set AR. This means that one can choose any set of parallel rules as active. Nevertheless, as mentioned before, the typical case is to choose $\mathcal{AR}$ as the set of all parallel rules composed of the rules of a set of units that interact with each other in the common environment. The rules of the considered unit may occur as single rules or as components of parallel rules in $\mathcal{AR}$.

2. All parallel and concurrent runs contain only derivations that apply active rules. Moreover, each such derivation is only accepted if its initial and result graph are allowed by the control condition. Moreover, each concurrent run contains at least one accepted parallel run. Hence, the set of concurrent runs is the quotient set of the parallel runs with respect to the true equivalence relation. This is reflected in the following observation the proof of which is straightforward and hence omitted.

Observation 1. Let $aut = (g, P, c)$ be an autonomous unit, let $\mathcal{AR} \subseteq \mathcal{R}_*$, and let $d = (G \overset{*}{\underset{\mathcal{AR}}{\Longrightarrow}} G')$ be a derivation. Then the following statements are equivalent.

1. $(G, G') \in SEM_{\mathcal{AR},P}(c)$
2. $d \in PAR_{\mathcal{AR}}(aut)$
3. $[d] \cap PAR_{\mathcal{AR}}(aut) \neq \emptyset$
4. $[d] \in CONCUR_{\mathcal{AR}}(aut)$

Examples

Two examples of autonomous units are depicted in Fig. 2. Both contain a single rule that – according to the control condition *free* – can be applied arbitrarily often. The goal of both units is equal to *all* which means that all parallel and concurrent runs of the units are successful.

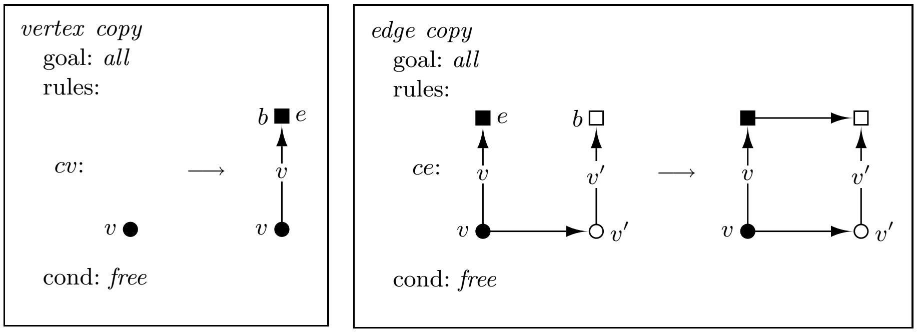

Fig. 2. The units *vertex copy* and *edge copy*

Given a directed edge-labeled graph G and a set V such that every node in G is a v-node, for some $v \in V$, the unit *vertex copy* on the left-hand side of Fig. 2 copies a node of G by generating a $\{b, e\}$-node and a v-labeled edge from the original node to its newly generated copy. The unit *edge copy* on the right side copies one unlabeled edge e' of G provided that $s(e')$ and $t(e')$ are already copied by two executions of *vertex copy.* It erases the e-loop at the copy of $s(e')$ and the b-loop at the copy of $t(e')$ in order to guarantee that b and e cannot be used by other applications of the rule *ce*. The label b indicates that the corresponding node is the beginning of a simple path and the label e indicates the end of such a path, respectively. Hence, b is removed if a new edge ends at that node, and e is removed if a new edge starts at that node. Multiple concurrent applications of *vertex copy* and *edge copy* generate simple paths from b-nodes to e-nodes that are copies of simple paths of G. Moreover, the units generate copies of simple cycles of G.

Applications of both rules depend on each other only in some cases. Concretely, an application of *ce* is causally dependent on the two copies of its nodes and on any edge copy that tries to use the same b- or e-loop. All other cases are independent so that all vertex copies can be done in parallel followed by all edge copies in parallel in the extreme case.

Autonomous units are meant to work within a community of autonomous units that modify the common environment together. Every community is composed of an overall goal that should be achieved, an environment specification that specifies the set of initial environments the community may start working with, and a set of autonomous units. The overall goal may be closely related to the goals of the autonomous units in the community. Typical examples are the goals admitting only graphs that satisfy the goals of one or all autonomous units in the community.

Definition 5 (Community). A *community* is a triple

$$COM = (Goal, Init, Aut),$$

where $Goal, Init \in \mathcal{X}$ are graph class expressions called the *overall goal* and the *initial environment specification*, respectively, and Aut is a set of autonomous units. The components of COM are denoted as $Goal_{COM}$, $Init_{COM}$, and Aut_{COM}, respectively.

In a community, all units act on the common environment in a self-controlled way by applying their rules. The active rules integrated in the semantics of autonomous units make it possible to define a concurrent semantics of a community in which every autonomous unit may perform its transformation processes. From the point of view of a single autonomous unit, the changes of the environment that are not caused by itself must be activities of the other units in the community. Hence, in every transformation step in a community, a multiset of the rules occurring in the autonomous units of the community is applied to the environment. All these multisets constitute the active rules of the community. This is reflected in the following definition.

Definition 6 (Active rules). Let $COM = (Goal, Init, Aut)$ be a community. Then the set of its *active rules* is defined by

$$\mathcal{AR}(COM) = (\bigcup_{aut \in Aut} P_{aut})_*.$$

Every concurrent run of a community must start with a graph specified as an initial environment of the community. Moreover, it must be a concurrent run of every autonomous unit participating in the community. Successful runs of communities are defined analogously to the successful runs of autonomous units.

Definition 7 (Concurrent community semantics).

1. Let $COM = (Goal, Init, Aut)$ be a community of autonomous units. Let $d_{aut} \in PAR_{\mathcal{AR}(COM)}(aut)$ for every $aut \in Aut$ with $d_{aut} \equiv d_{aut'}$ for all $aut, aut' \in Aut$. Let the common start graph of these equivalent derivations be in $SEM(Init)$. Then the common equivalence class is a concurrent run of COM.
2. The set of all concurrent runs of COM is denoted by $CONCUR(COM)$.
3. A concurrent run is *successful* if the common result graph is specified by $SEM(Goal)$.

As the definition of the community semantics shows, there is a strong connection between the semantics of a community $COM = (Goal, Init, Aut)$ and the semantics of an autonomous unit $aut \in Aut$. The concurrent semantics of COM is a subset of the semantics of *aut* with respect to the active rules of COM. Conversely, one may take the intersection of the concurrent runs of all autonomous units with respect to the active rules and restrict it to the derivations starting in an initial environment. Then one gets the concurrent semantics of the community. This reflects the autonomy because no unit can be forced to do anything that is not admitted by its own control. The following observation makes the described connection precise.

Observation 2. Let $COM = (Goal, Init, Aut)$ be a community. Then

$$CONCUR(COM) = \\ \{[G \underset{\mathcal{AR}(COM)}{\overset{*}{\Longrightarrow}} G'] \in \textstyle\bigcap_{aut \in Aut} CONCUR_{\mathcal{AR}(COM)}(aut) \mid G \in Init\}.$$

Proof. Let $d = (G \underset{\mathcal{AR}(COM)}{\overset{*}{\Longrightarrow}} G')$ be a derivation. Then by Definition 7, the class $[d]$ is in $CONCUR(COM)$ if and only if $G \in SEM(Init)$ and for each $aut \in Aut$, there is a derivation $d_{aut} \in PAR_{\mathcal{AR}(COM)}(aut)$ such that $d_{aut} \equiv d$, i.e., $[d_{aut}] = [d]$. By Definition 4, this means that $[d] \in \{[G \underset{\mathcal{AR}(COM)}{\overset{*}{\Longrightarrow}} G'] \in \bigcap_{aut \in Aut} CONCUR_{\mathcal{AR}(COM)}(aut) \mid G \in Init\}$.

If the control conditions satisfy certain properties, the preceding definitions establish a nice relation between the community semantics and the semantics of the single autonomous unit which is composed of the goal of the community, the union of all rules occurring in the autonomous units of the community and a control condition that, specifies the intersection of the semantics of the control conditions of the units. This is stated in the following observation.

Observation 3. Let $COM = (Goal, Init, Aut)$ be a community and let $union = (Goal, \bigcup_{aut\in Aut} P_{aut}, c)$ such that

$$SEM_{\mathcal{AR}(COM),P_{union}}(c) = \bigcap_{aut\in Aut} SEM_{\mathcal{AR}(COM),P_{aut}}(c_{aut}).$$

Then $CONCUR_{\mathcal{AR}(COM)}(union) = \bigcap_{aut\in Aut} CONCUR_{\mathcal{AR}(COM)}(aut)$.

Proof. Let $d = (G \underset{\mathcal{AR}(COM)}{\overset{*}{\Longrightarrow}} G')$ be a derivation. By Observation 1, we have that $[d] \in CONCUR_{\mathcal{AR}(COM)}(union)$ if and only if $(G, G') \in SEM_{\mathcal{AR}(COM),P}(c)$. By assumption, this is the case if and only if

$$(G, G') \in \bigcap_{aut\in Aut} SEM_{\mathcal{AR}(COM),P_{aut}}(c_{aut}),$$

and by Observation 1, this is equivalent to

$$[d] \in \bigcap_{aut\in Aut} CONCUR_{\mathcal{AR}(COM)}(aut).$$

It should be noted that the condition in Observation 3 can be satisfied if the following holds.

1. For each $aut \in Aut$ the semantics of the control condition c_{aut} does not depend on the parameter P_{aut}, i.e.,

$$SEM_{\mathcal{AR}(COM),P}(c_{aut}) = SEM_{\mathcal{AR}(COM),P'}(c_{aut}),$$

   for all $P, P' \subseteq \mathcal{R}$.
2. The class $\mathcal{C}$ is closed under intersection, i.e., for all $c, c' \in \mathcal{C}$ we have $c \wedge c' \in \mathcal{C}$ with $SEM_{\mathcal{AR},P}(c \wedge c') = SEM_{\mathcal{AR},P}(c) \cap SEM_{\mathcal{AR},P}(c')$.

In this case, the condition c of the unit *union* can be defined as $\bigwedge_{aut\in Aut} c_{aut}$.

Examples

The community *V-paths* shown in Fig. 3 solves a generalized form of the Hamiltonian path problem. More precisely, for some set $V = \{v_1, \ldots, v_r\}$, *V-paths* searches for copies of V-paths which are simple paths, that consist of r nodes and contain exactly one v_i-node for every $i = 1, \ldots, r$. Hence, these paths are Hamiltonian w.r.t. the set V and, consequently, if V coincides with the node set of the initial environment, the community solves the Hamiltonian path problem. This is done in a concurrent way with the four autonomous units *V-checker*, *vertex copy*, *edge copy*, and *check*. The initial component of the community specifies the set of *V-graphs* comprising all directed graphs with unlabeled edges and nodes, but where to each node a v-loop is added for some $v \in V$ The goal of *V-paths* is a subgraph condition specifying the set of all graphs that contain a *heureka*-node indicating that a V-path is found.

V-paths
goal: ▨ *heureka*
init: *V-graphs*
aut: *V-generator*, *vertex copy*, *edge copy*, *check*

Fig. 3. The community *V-paths*

The autonomous unit *V-checker* generates a set of $(V \cup \{check\})$-nodes where $check \notin V$. This node is used later on for analysing copies of simple paths. It is depicted in Fig. 4. The nodes generated by *V-checker* will be called checkers in the following.

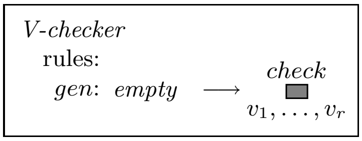

Fig. 4. The unit *V-checker*

The autonomous units *vertex copy* and *edge copy* are depicted in Fig. 2. As stated in Section 4, they copy simple paths of the initial graph and label their start nodes with a b-loop and their end nodes with an e-loop, each.

The unit *check* is depicted in Fig. 5 and contains the rules *start*, *go*, and *stop*. It searches for a copy of a simple path of the initial environment. The rule *start* begins the search at a b-node. It inserts a *go*-edge from the b-node to a $(V \cup \{check\})$-checker and deletes both the v-loop and the *check*-loop from the latter. The *check*-loop is deleted to avoid that the checker can be used for another path, and the v-loop is deleted in order to remember that the path has already passed through a copy of a v-node. The application of the rule *go* changes the source of the *go*-edge to the next node, say n, in the copy of a simple path. Moreover, it deletes the corresponding loop at the checker of the path, i.e., it deletes that loop at the target of the *go*-edge which has the same label as the loop of the node of which n is a copy. Hence, the rule *go* cannot move the source of the *go*-edge to an already visited node. The rule *stop* can be applied if there doesn't remain any labeled loop at the corresponding checker. Moreover the copy of the path ends which is indicated by the e-loop. The application of the rule deletes the *go*-edge and adds a *heureka*-loop to its target. Please note that the symbol λ at the checker in the left-hand side of the rule means that the checker has no labeled loop. Technically, this can be expressed by the rule with negative context condition (NC, L, K, R) where L consists of an e-node, a checker, and a *go*-edge from the e-node to the checker, NC consists of L plus a v-loop at the checker where $v \in V$, K is obtained from L by deleting the *go*-edge, and R is obtained from K by adding a *heureka*-loop to the checker.

The control condition is satisfied if the unit *check* applies in its last step a parallel rule composed of the rule *stop* only. Before the application of *stop*, parallel rules composed of *start* and *go* can be applied arbitrarily often. Hence, the unit finishes transforming graphs after the first application of the rule *stop*.

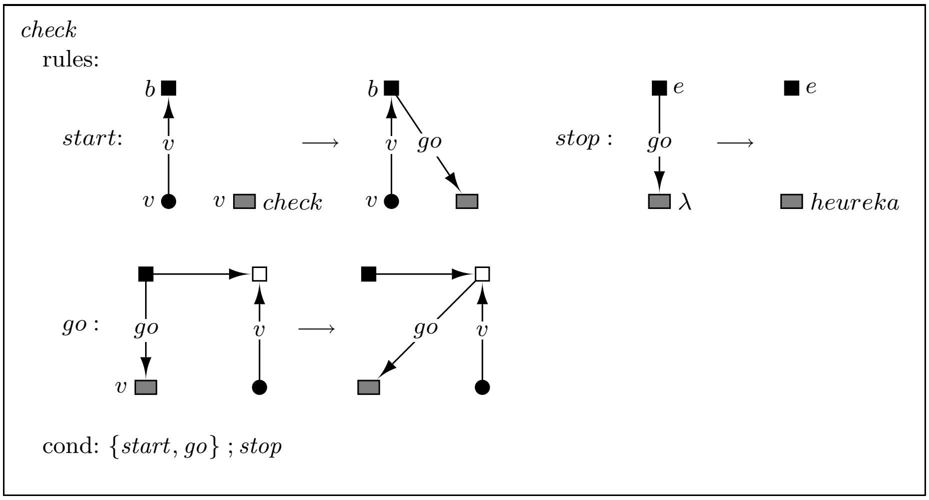

Fig. 5. The unit *check*

With respect to causal dependency, the following holds. The application of the rule of *V-checker* is independent of all other rule applications. The application of *start* must wait for the generation of its node copy and its checker. The rule *go* can be applied after a *start* application sequentially traversing a copy of a simple path. The rule *stop* is only applicable at the end of the path if at all. The applications of the *check*-rules are independent of each other if they concern different copies of simple paths. In particular, the check of a single copy of a simple path is never longer than r steps if r is the number of elements of V. It needs exactly r steps in the successful case of finding a V-path. Because of the control condition of the autonomous unit *check*, we get that all concurrent runs of the community *V-paths* are successful.

5 Canonical Derivations

The definition of concurrent runs of autonomous units reflects the principle of true concurrency meaning that the order of time of two rule applications is only fixed if the derivation steps are causally dependent. The disadvantage of the equivalence classes as concurrent runs is that they may contain an exponential number of equivalent derivations. This follows from the fact that a parallel derivation step applying n rules is equivalent to all iterated sequentializations including all $n!$ permutations of the n rule applications. There is a complete enumeration of each equivalence class starting with a parallel run in the class by iterated sequentializations and inverse sequentializations. But one may ask whether there is a more efficient method to check the equivalence of parallel runs. How and how fast equivalence can be checked, is often a fundamental question. In the case of concurrent runs of autonomous units, it is of particular interest because an equivalence class of parallel runs satisfies the control condition conditions if one member does. Therefore, the satisfaction of control conditions can only be checked up to equivalence.

In this section, we show that each equivalence class of parallel runs contains canonical derivations which are reduced forms with respect to a shift operator and which can be constructed from an arbitrary run by a quadratic number of shifts at most. A shift is a composition of a sequentialization followed by an inverse sequentialization moving some part of a parallel derivation step to the preceding step. In other words, one gets a quadratic equivalence check in this way if the canonical derivation is a unique representative of a concurrent run. Furthermore, it turns out that the canonical derivation is unique if the shift operator is confluent which applies in the DPO approach for example. For technical purposes, we also introduce the delay of a parallel derivation as the sum of the numbers of steps each atomic rule must wait before it is applied as well as the number of applied atomic rules.

The following notions and results can be found in [11] (cf. also [3]) for the DPO approach to graph transformation. They are adapted here to the case of concurrent runs in communities of autonomous units.

Definition 8 (Shift operator).

1. A parallel derivation $F \underset{p+q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H$ with $p \neq \mathbf{0} \neq q$ is the *shift* of the parallel derivation $F \underset{p}{\Longrightarrow} G \underset{q+r}{\Longrightarrow} H$ if there is a derivation $F \underset{p}{\Longrightarrow} G \underset{q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H$ such that the first two steps are the sequentialization of $F \underset{p+q}{\Longrightarrow} G'$ and the last two steps the sequentialization of $G \underset{q+r}{\Longrightarrow} H$.
2. The shift operator is closed under sequential composition of parallel derivations, i.e., a parallel derivation $E \overset{*}{\underset{P}{\Longrightarrow}} F \underset{p+q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H \overset{*}{\underset{P}{\Longrightarrow}} I$ is a *shift* of the parallel derivation $E \overset{*}{\underset{P}{\Longrightarrow}} F \underset{p}{\Longrightarrow} G' \underset{q+r}{\Longrightarrow} H \overset{*}{\underset{P}{\Longrightarrow}} I$ if $F \underset{p+q}{\Longrightarrow} G' \underset{r}{\Longrightarrow} H$ is a shift of $F \underset{p}{\Longrightarrow} G \underset{q+r}{\Longrightarrow} H$.
3. A parallel derivation is *canonical* if no shift is possible.
4. Let $s = (G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_m}{\Longrightarrow} G_m)$ be a parallel derivation with $r_i = \sum_{j=1}^{n_i} r_{ij}$, $r_{ij} \in \mathcal{R}, n_i \geq 1$ for $i = 1, \cdots, m$. Then the *delay* of s is defined by

$$delay(s) = \sum_{i=1}^{m} (i-1) \cdot n_i$$

and the *number of atomic rules* applied in s by $nar(s) = \sum_{i=1}^{m} n_i$.

Theorem 1. Let $s = (G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_m}{\Longrightarrow} G_m)$ be a parallel derivation with $r_i = \sum_{j=1}^{n_i} r_{ij}$, $r_{ij} \in \mathcal{R}, n_i \geq 1$ for $i = 1, \cdots, m$. Let s' be a shift of s. Then the following holds.

1. $s \equiv s'$.
2. $delay(s') < delay(s)$.
3. $0 \leq delay(s) \leq \frac{n\cdot(n-1)}{2}$ for $n = nar(s)$.
4. Let $s = s_0, s_1, \cdots, s_k$ be a sequence of parallel derivations such that s_i is a shift of s_{i-1} for $i = 1, \cdots, k$. Then $k \leq delay(s)$.
5. The equivalence class $[s]$ contains canonical derivations where some of them are obtained by iterating shifts as long as possible starting with s.
6. The canonical derivation in $[s]$ is unique if the shift operator is confluent, i.e., if s_1 and s_2 are shifts of some parallel derivation s_0, then there is a parallel derivation s_3 which is obtained from s_1 and s_2 by iterated shifts.
7. Let the canonical derivation in $[s]$ be unique. Then $s \equiv \overline{s}$ for some parallel derivation $\overline{s}$ if and only if iterated shifts as long as possible starting in s and $\overline{s}$ yield the same result.

Proof. 1. A shift is a composition of a sequentialization and an inverse sequentialization and yields equivalent derivations, therefore.
2. A shift moves at least one atomic rule to the preceding step such that it is delayed one step less while no rule must wait longer than before.
3. As the shift decreases the delay, a sequentialization increases it. Therefore, the worst case is a purely sequential derivation with one applied atomic rule per step. The delay of such a derivation is the sum of the first $n-1$ natural numbers which is $\frac{n\cdot(n-1)}{2}$.
4. As the delay cannot be negative and decreases with each shift, $delay(s)$ is an upper bound of the number of iterated shifts starting in s.
5. Accordingly, the iteration of shifts as long as possible terminates always yielding a canonical derivation.
6. It is well-known that a relation yields unique results by iterated application if it is confluent and terminating.
7. Let $\hat{s}$ be the result of the iterated shifts starting in s and $\overline{s}$. Then $s \equiv \hat{s} \equiv \overline{s}$ according to Point 1. Conversely, there is only one canonical derivation in $[s]$.

Examples

Let $[s]$ be a successful concurrent run of the community *V-paths*. Without loss of generality, one can assume that s is canonical (otherwise shifts as long as possible yield one). Due to the dependency analysis, all node copies and all checker generations are done in the first step and all edge copies and applications of *start* are done in the second step. The following $r-2$ steps are parallel applications of the rule *go* where r is the number of elements of the label set V. The last step consists of all parallel applications of the rule *stop* which is applied at least once because this is the only way to end successfully. This means that a successful run finds a V-path (and hence a Hamiltonian path as a special one) in $r+1$ steps.

If enough node and edge copies are made in a fair way, then one gets copies of all simple paths. If enough checkers are generated, then it can be checked whether there is a V-path among all simple paths. In other words, the concurrent semantics of *V-paths* can solve the V-path problem in a linear number of steps with high probability depending on the number of copies. To make this true, one must assure that concurrent steps are not delayed for too long. The canonical derivation does the job because all possible rule applications are performed as early as possible. But less strict runs will also work if the number of shifts that transform them into the canonical derivation is small.

This result is quite significant as it shows that an *NP*-complete problem can be solved by a parallel run in a polynomial number of steps. This holds for concurrent runs, too, if there is not much unnecessary delay. Clearly, it is well-known that parallelism is a way to overcome the *P*=*NP*-problem. The message here is that autonomous units do the job if they are many enough and interact properly. To keep the example simple, we do not employ any kind of heuristics. Nevertheless, autonomous units are suitable candidates to model heuristic methods (see e.g. [20,15].)

6 Related Work

The present investigation is mainly related to work in three areas. With respect to the concurrent semantics, it contributes to the *theory of concurrency*. Petri nets are shown to be special cases of communities of autonomous units (cf. [9]). The relation to other models of concurrent processes is still an open research topic.

Concerning autonomy, our approach is closely related to *multiagent systems*. In [19], it is sketched that communities of autonomous units are a kind of rule-based realization of the axiomatic definition of multiagent systems in the sense of Wooldridge and others (see,e.g., [21]). How our concept is related to other graph-transformational approaches to agent and actor systems like [10,5,4], should be investigated in the future.

Last but not least, communities of autonomous units are devices to model *interactive processes and distributed systems*. Within the area of graph transformation, the approach is closely related to Manfred Nagl's and others' work on the IPSEN Project and the IMPROVE Project (see, e.g., [16,17]). While we start from a theoretical base and try to expand the concepts to reach practical use, the Nagl school is rooted in the software engineering point of view from the beginning. It would be of great interest to investigate the relations in more details because quite some synergy may emerge from a common framework.

7 Conclusion

This is the third paper on the semantics of autonomous units. After the sequential semantics in [8] and the parallel semantics in [13], we have introduced the concurrent semantics based on the idea of true concurrency. A concurrent run is an equivalence class of parallel runs w.r.t. the true-concurrency equivalence. It can be represented by canonical derivations where a derivation is canonical if no shift is possible meaning that each rule application is in the first step or causally dependent of the preceding step. The example shows that an *NP*-hard problem can be solved by linear concurrent runs of a community of autonomous units where a concurrent run is linear if it contains a parallel run of linear length.

The paper provides the very first investigation of the concurrent semantics of autonomous units. Further studies are needed to get a better insight into the matter. This includes a thorough comparison with other approaches to concurrency like communicating sequential processes, calculus of communicating systems, traces, bigraphs, etc. W.r.t. Petri nets, the relation is already quite clear. In [13], it is shown that a place/transition system can be seen as a community of autonomous units where each transition is the single rule of a unit and the firing of a multiset of transitions satisfies the true-concurrency condition so that these communities of place/transition systems fit into the framework of this paper. Moreover, it should be thoroughly studied what are of the consequences of requiring that an equivalence class must contain a single successful run.

References

1. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A., Taentzer, G.: Graph transformation for specification and programming. Science of Computer Programming 34(1), 1–54 (1999)
2. Baldan, P., Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Rossi, F.: Concurrent semantics of algebraic graph transformations. In: 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, pp. 107–185. World Scientific, Singapore (1999)
3. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In: Rozenberg [18], pp. 163–245
4. Depke, R., Heckel, R.: Modeling and analysis of agents' goal-driven behavior using graph transformation. In: Ryan, M.D., Meyer, J.-J.C., Ehrich, H.-D. (eds.) Objects, Agents, and Features. LNCS, vol. 2975, pp. 81–97. Springer, Heidelberg (2004)
5. Depke, R., Heckel, R., Küster, J.: Formal agent-oriented modeling with graph transformation. Science of Computer Programming 44, 229–252 (2002)
6. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
7. Hölscher, K., Klempien-Hinrichs, R., Knirsch, P., Kreowski, H.-J., Kuske, S.: Autonomous units: Basic concepts and semantic foundation. In: Hülsmann, M., Windt, K. (eds.) Understanding Autonomous Cooperation and Control in Logistics – The Impact on Management, Information and Communication and Material Flow, pp. 103–120. Springer, Heidelberg (2007)

8. Hölscher, K., Kreowski, H.-J., Kuske, S.: Autonomous units and their semantics — the sequential case. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 245–259. Springer, Heidelberg (2006)
9. Hölscher, K., Kreowski, H.-J., Kuske, S.: Autonomous units to model interacting sequential and parallel processes. Fundamenta Informaticae 92(3), 233–257 (2009)
10. Janssens, D.: Actor grammars and local actions. In: 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, pp. 57–106. World Scientific, Singapore (1999)
11. Kreowski, H.-J.: Manipulationen von Graphmanipulationen. Ph.D. thesis, Technische Universität Berlin (1977)
12. Kreowski, H.-J., Kuske, S.: Graph transformation units with interleaving semantics. Formal Aspects of Computing 11(6), 690–723 (1999)
13. Kreowski, H.-J., Kuske, S.: Autonomous units and their semantics - the parallel case. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 56–73. Springer, Heidelberg (2007)
14. Kuske, S.: More about control conditions for transformation units. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 323–337. Springer, Heidelberg (2000)
15. Kuske, S., Luderer, M.: Autonomous units for solving the capacitated vehicle routing problem based on ant colony optimization. Electronic Communications of the EASST 26, 23 pages (2010)
16. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)
17. Nagl, M., Marquardt, W. (eds.): Collaborative and Distributed Chemical Engineering: From Understanding to Substantial Design Process Support - Results of the IMPROVE Project. LNCS, vol. 4970. Springer, Heidelberg (2008)
18. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific, Singapore (1997)
19. Timm, I., Kreowski, H.-J., Knirsch, P., Timm-Giel, A.: Autonomy in software systems. In: Hülsmann, M., Windt, K. (eds.) Understanding Autonomous Cooperation and Control in Logistics – The Impact on Management, Information and Communication and Material Flow, pp. 255–274. Springer, Heidelberg (2007)
20. Tönnies, H.: An evolutionary graph transformation system as a modelling framework for evolutionary algorithms. In: Mertsching, B., Hund, M., Aziz, Z. (eds.) KI 2009. LNCS, vol. 5803, pp. 201–208. Springer, Heidelberg (2009)
21. Wooldridge, M., Jennings, N.R.: Intelligent agents: Theory and practice. The Knowledge Engineering Review 10(2) (1995)