*Fundamenta Informaticae XX (2009) 1–25*
*IOS Press*


# Autonomous Units to Model Interacting Sequential and Parallel Processes

**Karsten Hölscher***

**Hans-Jörg Kreowski***

**Sabine Kuske***

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

**Abstract.** In this paper, we introduce the notion of a community of autonomous units as a rule-based and graph-transformational device to model processes that run interactively but independently of each other in a common environment. The main components of an autonomous unit are a set of rules, a control condition, and a goal. Every autonomous unit transforms graphs by applying its rules so that the control condition is satisfied. If the goal is reached the resulting transformation process is successful. A community contains a set of autonomous units, an initial environment specification, and an overall goal. In every transformation process of a community the autonomous units interact via their common environment. As an example, the game Ludo is modeled as a community of self-controlled players who interact on a common board. The emphasis of the presented approach is laid on the study of the formal semantics of a community as a whole and of each of its member units separately. In particular, a sequential as well as a parallel semantics is introduced, and communities with parallel semantics are compared with Petri nets, cellular automata, and multiagent systems.

**Keywords:** Autonomous units, graph transformation, formal semantics

*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).

# 1. Introduction

Data processing of today (like communication networks, multiagent systems, swarm intelligence, ubiquitous, wearable and mobile computing) is often distributed and comprises various components that run partially independent of each other, but may access and update the same information structures, communicate with each other and interact in various ways. They may cooperate to reach a common goal or may compete with each other to achieve their individual aims. Typical examples of this kind are logistic processes and systems like transport and production networks where many actors from different companies come together and cooperate to a certain degree. But they are usually still competitors who are not willing to transfer their control to others or to a central entity. On the more technical level, transport networks, for example, comprise many transport vehicles, lots of goods to be shipped, various further components for storing, loading, reloading, etc. It is not meaningful to model such a network as a centralized system with a single control. The same applies to production networks with respect to the involved machines, materials, storage areas, etc.

The main idea of this paper is to provide a formal transformational and rule-based framework for the modeling of such systems composed of a variety of highly self-controlled components that make their decisions on their own depending on the information they get from their environment.

The basic notion is that of a community of autonomous units which exist in a common environment. There are initial environments to start computational processes, and there is an overall goal. Each autonomous unit in a community has its own individual goal in addition. To reach its goal, the autonomous unit can apply its rules or use so-called transformation units in order to describe more complex transformations than rules can do. Moreover, each autonomous unit has a control mechanism to decide which rule or transformation unit is applied next. This establishes the autonomy of a unit.

The autonomous units in a community are not directly aware of each other, but they may notice the outcome of the activities of their co-units because some of their rules may become applicable and others may loose this possibility. In this way, autonomous units can communicate and interact. To cover these phenomena in the process semantics of a single autonomous unit, we assume a change relation on environments that makes the environment dynamic.

In the first part of the paper, we introduce the sequential semantics of communities of autonomous units. It is given by all sequential processes - finite and infinite - that start in an initial environment, are composed of rule applications and applications transformation units, and follow, in each step, the control of the active autonomous unit. From the point of view of a single autonomous unit, this means that its own actions (being rule applications or applications of transformation units) take place interleaved with other changes of the dynamic environment caused by the coexisting autonomous units.

Clearly, the sequential semantics is only adequate if one deals with systems in which activities take place one after the other. Examples of this kind are card and board games, sequential algorithms, single-processor systems and such. Moreover, there are many modeling approaches the semantics of which assumes one action at a time. But even sequential systems may consist of self-controlling components that decide about their own activities independently of the others like the examples of card and board games with several players show.

To cover parallelism, we assume that not only single rules but multisets of rules can be applied to environments. This means that in each step many rules can be applied and single rules multiple times. As the rules may belong to different autonomous units, the autonomous units act in parallel. A parallel process of a single autonomous unit can be described as a sequence of application of multisets of rules

of this autonomous unit in parallel with changes of the environment.

The concept of autonomous units is approach-independent in the sense that the classes of environments, rules, control conditions, and goals are not determined in advance but can be provided by a transformation approach that is plugged in the community where the autonomous units interact. Nevertheless, in this paper, environments are assumed to be graphs because graphs are very generic and allow to model the states of a wide spectrum of rule-based systems and, in particular, of all processes that can be modeled with any of the existing graph transformation approaches.

The benefit we expect of using autonomous units is to obtain a general, easy-to-use, and visually well-understandable formal framework with a precise semantics that allows to model systems of interacting components so that on the one hand external control structures are set aside and on the other hand string-based representation is replaced by graph- and rule-based representation that allows to visualize and specify the systems more like they are.

The introduction and investigation of autonomous units is mainly motivated by the Collaborative Research Centre 637 *Autonomous Cooperating Logistic Processes*. This interdisciplinary project focuses on the question whether logistic processes with autonomous control may be more advantageous than those with central control, especially regarding time, costs and robustness. The guiding principle of autonomous units is the integration of autonomous control into rule-based models of processes. The aims are

1. to describe algorithmic and particularly logistic processes in a very general and uniform way, based on a well-founded semantic framework,

2. to provide a range of applications that reaches from classical process chain models like the ones by Kuhn (see, e.g., [11]) or Scheer (see, e.g., [15]) and the well-known Petri nets (see, e.g., [13, 1]) to agent systems see, e.g., [17]) and swarm intelligence (see, e.g., [8]),

3. to comprise the foundation of the dynamics of processes by means of rules where rule applications define process, transformation, and computation steps yielding local changes.

The paper is organized as follows. In Sect. 2 we recall the notion of a graph transformation approach. In Sect. 3 autonomous units are introduced and a sequential semantics for them is given. Sect. 4 presents communities of autonomous units and formalizes their sequential semantics. In Sect. 5 we present a case study modeling the players of the board game *Ludo* as autonomous units. Sect. 6 introcuces a formal semantics for the parallel case. To shed some first light on the significance and usefulness of communities of autonomous units with parallel-process semantics, we compare our concepts with the parallelism provided by other well-known frameworks in Sections 7-9. In particular, Sect. 7 translates place/transition systems into communities of autonomous units and shows that firing sequences of multisets of transitions correspond to parallel processes of the associated community. Similarly, cellular automata can be considered as communities of autonomous units as shown in Sect. 8. Cellular automata are particularly interesting as all their cells change states simultaneously so that the mode of computation is massively parallel. In Sect. 9, we discuss the relationship between communities of autonomous units and multiagent systems. As the latter are defined in an axiomatic way, the former can be seen as rule-based models providing an operational semantics for multiagent systems independently of the implementation of agents. Sect. 10 concludes the paper.

Preliminary short versions of parts of this paper are published in [7, 10]. The basic ideas are sketched in [6].

## 2. Graph transformation approaches

Whenever one has to do with dynamic graph-like structures, graph transformation (see also [14]) constitutes an adequate formal specification technique because it supports the visual and rule-based transformation of such structures in an intuitive and direct way. The ingredients of graph transformation are provided by a so-called graph transformation approach. In this section, we recall the notion of a graph transformation approach as introduced in [9] but modified with respect to the class of control conditions.

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 for example sets of initial and terminal graphs of graph transformation processes.

**Definition 2.1. (Graph transformation approach)**
A graph transformation approach is a system $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \mathcal{X}, \mathcal{C})$ the components of which are defined as follows.

- $\mathcal{G}$ 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 $SEM(r) \subseteq \mathcal{G} \times \mathcal{G}$.
- $\mathcal{X}$ is a class of *graph class expressions* such that each $x \in \mathcal{X}$ specifies a set of graphs $SEM(x) \subseteq \mathcal{G}$.
- $\mathcal{C}$ is a class of *control conditions* such that each $c \in \mathcal{C}$ specifies a set of sequences $SEM_{Change}(c) \subseteq SEQ(\mathcal{G})$ where $Change \subseteq \mathcal{G} \times \mathcal{G}$. [1]

**Remark.** The relation $Change$ describes the changes that can occur in the environment of an autonomous unit. Hence, control conditions have a loose semantics which depends on the changes of the environment given by $Change$.

### 2.1. Examples.

In the following we present some instances of the components of graph transformation approaches. They are used in the following sections. Further examples of graph transformation approaches can be found in, e.g., [14].

**Graphs.** A well-known instance for the class $\mathcal{G}$ is the class of all directed edge-labeled graphs. Such a graph is a system $G = (V, E, s, t, l)$ where $V$ is a set of nodes, $E$ is a set of edges, $s, t\colon E \to V$ 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 $V_G$, $E_G$, $s_G$, $t_G$, and $l_G$, respectively. 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

[1] $SEQ(\mathcal{G})$ denotes the set of finite and infinite sequences over $\mathcal{G}$.

of $s_G$, $t_G$, and $l_G$ to $E_M$. A graph morphism $g\colon L \to G$ from a graph $L$ to a graph $G$ consists of two mappings $g_V : V_L \to V_G$, $g_E\colon E_L \to E_G$ 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 $l_G(g_E(e)) = l_L(e)$. In the following we omit the subscript $V$ or $E$ of $g$ if it can be derived from the context.

Other classes of graphs are trees, forests, Petri nets, undirected graphs, hypergraphs, etc.

**Rules.** As a concrete example of rules we consider the so-called double-pushout rules [3] each of which consists of a triple $r = (L, K, R)$ of graphs such that $L \supseteq K \subseteq R$. Graph transformation rules can be depicted in several forms. In the following they are either shown in the form $L \supseteq K \subseteq R$ or 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 different nodes of $K$ are distinguished by different forms and fill-styles.

The application of a rule to a graph $G$ yields a graph $G'$, if one proceeds according to the following steps: (1) Choose a graph morphism $g\colon L \to G$ so that for all items $x, y$ (nodes or edges) of $L$ $g(x) = g(y)$ 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.) (3) Add $R$ to the resulting graph $D$, and (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.

A graph transformation rule $(L, K, R)$ with positive context is a quadruple $(PC, L, K, R)$ such that $L \subseteq PC$. It can be applied to $G$ by applying $(L, K, R)$ to $G$ as described provided that there is a morphism $g'\colon PC \to G$ such that the restriction of $g'$ to $L$ equals $g$. In the following, a rule with positive context is depicted as $PC \supseteq L \supseteq K \subseteq R$ where different fill-styles determine the nodes and edges of $L$ in $PC$.

**Graph class expressions.** Every subset $M \subseteq \mathcal{G}$ is a graph class expression that specifies itself, i.e., $SEM(M) = M$. 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 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. Another example of a graph class expression is a *subgraph condition*, i.e., a graph $G$ that admits all graphs that have (an isomorphic copy of) $G$ as subgraph. 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 parallel graph transformations, i.e. $SEM_{Change}(free) = SEQ(\mathcal{G})$ for all $Change \subseteq \mathcal{G} \times \mathcal{G}$. Another useful control condition is $alap(P)$ where $P \subseteq \mathcal{R}$. It applies $P$ as long as possible. More precisely, for every $Change \subseteq \mathcal{G} \times \mathcal{G}$, $SEM_{Change}(alap(P))$ consists of all finite sequences $(G_0, \ldots, G_n) \in SEQ(\mathcal{G})$ for which there is an $i \in \{0, \ldots, n\}$ such that no rule in $P$ can be applied to the graphs in $(G_i, \ldots, G_n)$. The condition $alap(P)$ can also be used to specify infinite sequences, a more complicated case that is not needed here.

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

The notion of a graph transformation approach is very general and requires just what is needed to introduce autonomous units and their sequential and parallel semantics in Sections 3, 4 and 6. Graphs are assumed to be items to which rules can be applied, and rules are assumed to provide a binary relation

on graphs. From this point of view, graphs could be called configurations, environments, objects or something like this, and rules could be called operations, events or actions. That we speak about graphs and rules, has the following reasons.

1. In the graph transformation literature one encounters quite a variety of different notions of graphs, rules, rule applications, and regulations of rule applications which have some basic properties in common. Our notion of a graph transformation approach is intended to cover most of these features in a unifying framework.

2. The notion of a graph is very generic. There are many variants. All kinds of diagrams can be seen as graphs. And many other structures can be nicely represented as graphs so that the requirement of a class of graphs is not very exclusive and reflects the genericity of the concept.

3. In all the examples we have considered so far, the environments of units are or can be considered as graphs. The examples of this paper, the Ludo game, Petri nets, and cellular automata, are quite typical in this respect.

4. The study of autonomous units is still at the beginning. Further investigations may demand further properties of graphs and rules which will be added to the notion of a graph transformation approach whenever needed. Properties to be expected later on are some distinctions between nodes and edges or some locality of rule application, for example.

## 3. Autonomous units with sequential semantics

Autonomous units act and interact in a common environment which is modeled as a graph. An autonomous unit consists basically of a set of graph transformation rules, a control condition, and a goal. The graph transformation rules of an autonomous unit $aut$ specify the transformations the unit $aut$ can perform. Such a transformation can be for example a movement of the autonomous unit within the current environment, the exchange of information with other autonomous 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 or infinitely often. In this first approach the goal of an autonomous unit is a graph class expression determining how the transformed graphs should look like, eventually.

In practice, autonomous units may also want to execute atomic environment transformations that cannot be specified with a mere graph transformation rule but with a set of rules that should be applied in a specific order. For this purpose we use the concept of transformation units introduced in e.g. [9].In more detail, transformation units encapsulate sets of rules and control components that specify binary relations on graphs. Moreover, they allow to structure large rule sets hierarchically into smaller pieces because they provide an import component. As rules, transformation units specify binary relations on graphs. The application of a transformation unit via an autonomous unit cannot be interrupted by environment changes executed by other autonomous units or by local rules of the autonomous unit.

Transformation units are inductively defined over their depth where a transformation unit has depth zero if it does not use any other transformation unit. Otherwise its depth is the maximum of the depths of all imported units plus one.

**Definition 3.1. (Transformation unit)**

1. An *transformation unit* of *depth* 0 is a system $tu = (\emptyset, P, C)$ where $P \subseteq \mathcal{R}$ is a set of graph transformation rules, and $C$ is a control component with $SEM(C) \subseteq \mathcal{G} \times \mathcal{G}$. The depth of $tu$ is denoted by $depth(tu)$.

2. Let $U \neq \emptyset$ be a set of transformation units and let $n = \max\{depth(u) \mid u \in U\}$ where $depth(u)$ denotes the depth of the unit $u$. Then a *transformation unit* of *depth* $n + 1$ is a system $tu = (U, P, C)$ where $P$ and $C$ are defined as in point one of this definition.

3. The components of $tu$ are also denoted by $U_{tu}$, $P_{tu}$, and $C_{tu}$, respectively.

**Remarks.**

1. In [9], the component $C$ of Definition 3.1 is called *control condition* but it is different from the control conditions in the graph transformation approaches defined in Section 2 because it specifies a binary relation and not a set of sequences. A typical example of a control component for transformation units is a finite automaton the edges of which are labeled with rules and imported transformation units. It specifies all pairs of graphs $(G, G')$ that can be obtained by applying the rules and transformation units on a path from the initial state of the automaton to a final state (in the same order as the edges in the path are visited) starting with $G$ and ending with $G'$. In general, the control components of the above definition can be regarded as a specific case of the control conditions of Section 2 because every pair $(G, G')$ allowed by a control component of a transformation unit can be regarded as the set of all finite sequences from $G$ to $G'$.

2. The transformation units in [9] contain also graph class expressions in order to specify start and end graphs of graph transformations. This specific feature of transformation units is not needed in the following. However, it is worth noting that graph class expressions can be represented as control components in a straightforward way, so that every transformation unit of [9] can be easily translated into a transformation unit the graph class expressions of which specify the class of all graphs. Hence, the above definition does not really restrict the concept of transformation units.

As in [9] we concentrate here on a sequemtial semantics of transformation units that contains all pairs of graphs $(G, G')$ so that $G'$ can be obtained from $G$ via the successive application of local rules and imported transformation units and $(G, G')$ is specified by $C$.

**Definition 3.2. (Semantics of transformation units)**
Let $tu = (U, P, C)$ be a transformation unit. Let $(G_0, \ldots, G_n) \in SEQ(\mathcal{G})$. Then $(G_0, G_n) \in SEM(tu)$ if

- there is a sequence $(x_1, \ldots, x_n) \in SEQ(U \cup P)$ such that for $i = 1, \ldots, n$

  $$(G_{i-1}, G_i) \in SEM(x_i),$$

  and

- $(G_0, G_n) \in SEM(C)$.

It is worth noting that the semantics of autonomous units is inductively defined meaning that it covers the case where no transformation unit is imported and in the case where the set of imported transformation units is not empty the semantics of every imported transformation unit is recursively computed.

Now autonomous units can be defined.

**Definition 3.3. (Autonomous unit)**
An *autonomous unit* is a system $aut = (g, U, P, c)$ where $g \in \mathcal{X}$ is the *goal*, $U$ is a set of transformation units, $P \subseteq \mathcal{R}$ is a set of graph transformation rules, and $c \in \mathcal{C}$ is a control condition. The components of $aut$ are also denoted by $g_{aut}$, $U_{aut}$, $P_{aut}$, and $c_{aut}$, respectively.

Every autonomous unit induces a set of atomic (i.e. not interruptible) environment transformations that consist of the semantic relation of all local rules plus the relations given by its transformation units.

**Definition 3.4. (Atomic transformations)**
The set of *atomic transformations* of an autonomous transformation unit $aut = (g, U, P, c)$ is defined as $AT(aut) = \bigcup_{x \in U \cup P} SEM(x)$ .

An autonomous unit modifies an underlying environment while striving for its goal. Its semantics consists of a set of transformation processes being finite or infinite sequences of environment transformations. An environment transformation is the application of a local rule, the invocation of a transformation unit, or an environment change performed by some other autonomous unit that is working in the same environment. These environment changes are given as a binary relation on environments. Hence, in this sequential approach a transformation process of an autonomous unit interleaves local rule applications and applications of used transformation units with environment changes specified by other components.

Autonomous units regulate their transformation processes by choosing in every step only those rules and transformation units that are allowed by its control condition. A finite transformation process is called successful if its last environment satisfies the goal of the autonomous unit. Every infinite transformation process is successful if it contains infinitely many environments that satisfy the goal.

**Definition 3.5. (Sequential semantics of autonomous units)**

1. Let $aut = (g, U, P, c)$ be an autonomous unit and let $Change \subseteq \mathcal{G} \times \mathcal{G}$. Let $s = (G_0, G_1, \ldots) \in SEQ(\mathcal{G})$. Then $s \in SEM_{Change}(aut)$ if

   - for $i = 1, \ldots, |s| - 1$ if $s$ is finite [2] and for $i \in \mathbb{N}^+$ if $s$ is infinite

     $$(G_{i-1}, G_i) \in AT(aut) \cup Change$$

   - $s \in SEM_{Change}(c)$.

2. The sequence $s$ is called a *successful transformation process* if $s$ is finite and $G_{|s|-1} \in SEM(g)$ or if $s$ is infinite and there is an infinite monotone sequence $i_0 < i_1 < i_2 < \cdots$ with $G_{i_j} \in SEM(g)$ for all $j \in \mathbb{N}$.

[2] For $s = (G_0, \ldots, G_n)$ the length $n + 1$ of $s$ is denoted by $|s|$.

# 4. Communities of autonomous units

Autonomous units are meant to work within a community of autonomous units that modify the common environment together. In the sequential case these modifications take place in an interleaving manner. 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 successful semantic sequences w.r.t. one or all autonomous units in the community.

**Definition 4.1. (Community)**
A *community* is a triple $CAU = (Goal, Init, Aut)$, 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.

In a community all autonomous units work in a self-controlled way by applying their rules or their transformation units to the common environment. The change relation integrated in the semantics of autonomous units makes it possible to define an interleaving semantics of a community in which every autonomous unit may perform its transformation processes. For this purpose it is necessary to define for every autonomous unit the set of atomic transformations of all other autonomous units in the community.

**Definition 4.2. (Change relation)**
Let $CAU = (Goal, Init, Aut)$ be a community. Then for each $aut \in Aut$ the *change relation* w.r.t. $aut$ is defined as $Change(aut) = \bigcup_{aut' \in Aut - \{aut\}} AT(aut')$.

Every transformation process of a community must start with a graph specified as an initial environment of the community. Moreover, it must be in the sequential semantics of every autonomous unit participating in the community. Analogously to successful transformation processes of autonomous units, a finite transformation process of a community is successful if its last environment satisfies the overall goal. Every infinite transformation process of a community is successful if it meets infinitely many environments that satisfy the overall goal.

**Definition 4.3. (Sequential community semantics)**

1. Let $CAU = (Goal, Init, Aut)$. Then the *sequential community semantics* of $CAU$, denoted by $SEM(CAU)$, consists of all finite or infinite sequences $s = (G_0, G_1, \ldots) \in SEQ(\mathcal{G})$ such that $G_0 \in SEM(Init)$ and $s \in SEM_{Change(aut)}(aut)$ for all $aut \in Aut$.

2. The sequence $s$ is a *successful transformation process* if $s$ is finite and $G_{|s|-1} \in SEM(Goal)$ or or there is an infinite monotone sequence $i_0 < i_1 < i_2 < \cdots$ with $G_{i_j} \in SEM(g)$ for all $j \in \mathbb{N}$.

**Remarks.**

1. As the definition of the community semantics shows, there is a strong connection between the semantics of a community $CAU = (Goal, Init, Aut)$ and the semantics of an autonomous unit $aut \in Aut$. More precisely, the semantics of $CAU$ is a subset of the semantics of $aut$ w.r.t. the change relation $Change(aut)$. Formally, this means that $SEM(CAU) \subseteq SEM_{Change(aut)}(aut)$ for all $aut \in Aut$.

2. One may take the intersection of the sequential semantics of all autonomous units with respect to their own change relation and restrict it to the sequences starting in an initial environment. Then one gets the sequential 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. To put it in another way, every transformation process of a community is a subset of the transformation processes obtained by the intersection of the semantics of all autonomous units in the community, i.e., $SEM(CAU) \subseteq \bigcap_{aut \in Aut} SEM_{Change(aut)}(aut)$. In the case where the initial environment expression specifies the class of all graphs, we get the equality, i.e., $SEM(CAU) = \bigcap_{aut \in Aut} SEM_{Change(aut)}(aut)$

3. Obviously, only atomic transformations of the participating autonomous units are applied in every transformation process, i.e. for every $s = (G_0, G_1, \ldots) \in SEM(CAU)$ we have $(G_{i-1}, G_i) \in \bigcup_{aut \in Aut} AT(aut)$ where $i = 1, \ldots, |s| - 1$ if $s$ is finite and $i \in \mathbb{N}^+$ if $s$ is infinite. The proof is straightforward: By Def. 4.3 we have that $s \in SEM_{Change(aut)}(aut)$ for all $aut \in Aut$. Let $aut \in Aut$. Then by Def. 3.5 we get for $i = 1, \ldots |s| - 1$ if $s$ is finite and for $i \in \mathbb{N}^+$ if $s$ is infinite that $(G_{i-1}, G_i) \in AT(aut) \cup Change(aut)$. By Def. 4.2 $(G_{i-1}, G_i) \in AT(aut) \cup \bigcup_{aut' \in Aut - \{aut\}} AT(aut')$. This implies that $(G_{i-1}, G_i) \in \bigcup_{aut \in Aut} AT(aut)$.

## 5. Modeling Ludo players as autonomous units

Board games are a typical example of communities of autonomous units with sequential semantics where the board provides the common environment and the players are the autonomous units. As a concrete example we consider in this section the game *Ludo*. [3]

The graph transformation approach used in this example consists of labeled directed graphs and double-pushout rules (cf. [3]). The control conditions used are regular expressions and priorities. As graph class expressions we use subgraph conditions and the graph class expression $all$ specifying the class of all graphs.

A possible environment graph of *Ludo* is the initial game situation where four players of different colors have all their tokens at the start place and there is one die showing an arbitrary number between one and six. This graph is depicted in Fig. 1. Every player is drawn as a kind of actor labeled with a color out of $b$(lue), $y$(ellow), $r$(ed), and $g$(reen) so that every player has a different color. Technically, a player is a labeled node. The players are connected via some directed edges indicating the playing succession. The game board consists of a start node and four home nodes for every player and a set of round nodes. The start node of a $c$-labeled player is depicted as a $c$-labeled hexagon. The home nodes which are drawn as rhombuses are labeled with the one-element set $\{c\}$. Every $c$-labeled player has four $c$-labeled tokens that all sit at her/his start node at the beginning of a game. The fact that a token of color $c$ is sitting at a node $v$ is visualized with a $c$-labeled token that is connected to $v$ via an undirected edge. Technically, this can be modeled by means of a $c$-labeled loop connected to the node $v$. The directed edges between the nodes of the game board indicate where and in which direction the tokens can move around the game board.

Every round node and every directed edge between round and home nodes are labeled with a set $M \subseteq \{b, y, g, r\}$. The label of every round node contains all colors that can visit this node. Since at the beginning of a game all round nodes are vacant, i.e. they can be visited by all colors, they are all labeled

[3] There exist several distinct versions of the game *Ludo*. In this paper we consider one of the standard German versions.

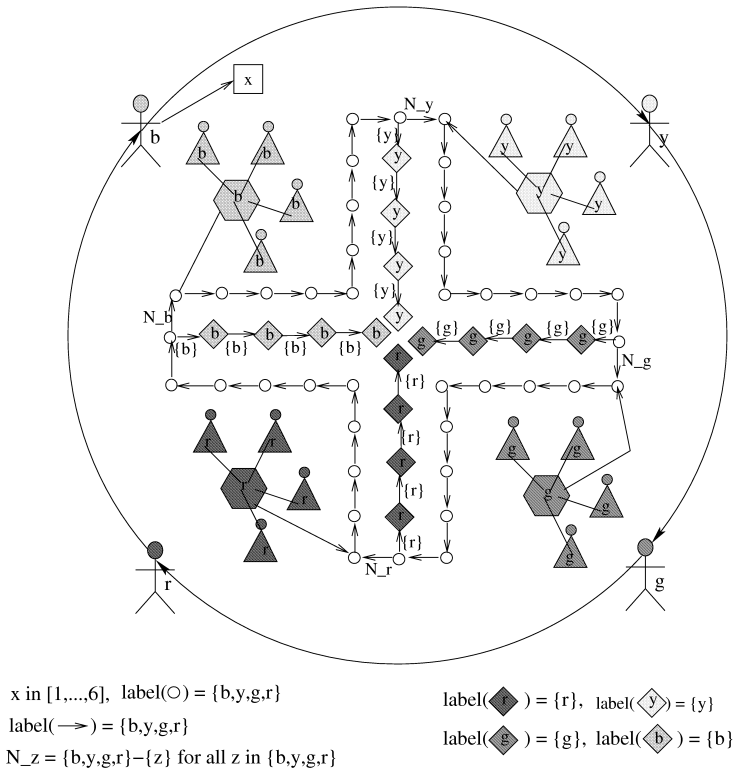


Figure 1. An environment of *Ludo*

with $\{b, y, g, r\}$. The labels of the edges connecting home and round nodes contain also all colors the tokens of which can move via these edges. For example, only yellow tokens can move to a home node of a yellow player. Moreover, no yellow token may go over the edge labeled with $N\_y = \{b, g, r\}$, because it has to enter its home. Please note that the labels of most of the nodes and edges of Fig. 1 are depicted below the graph in order to keep the graph easy to read.

The goal of every player is to have all four tokens at home, one in each home node. To reach a home place, a token must go from the start place over the round fields in the indicated direction. To move a token, a die must be thrown. If a six is thrown the current player must move one of her/his tokens from the start node to the first round node, i.e. to the round node connected to the start node. If there is no token left at the start node, the player can take any other of her/his tokens. A six allows for throwing again. We assume here that the blue player starts to play. This is why the $b$-labeled player is holding the die (represented by the edge from the player to the die). Afterwards it is the turn of the yellow player.

Every player of *Ludo* can be realized as the autonomous unit depicted in Fig. 2. The goal of a player $c$ is the subgraph condition consisting of the home of $c$ in which every node is connected to a $c$-labeled token. The rules and the transformation units model all possible actions of a player. The possible values of the variables occurring in the left- or right-hand side are put under the arrow. If the label of a node or an edge is not significant it is omitted in the rule, i.e. an item without a label can be matched to an item with any label. The rule *go-to-startpoint* of the autonomous unit *player* moves a token that has been kicked out to its home node. As the control condition prescribes this rule has the highest priority, i.e. it

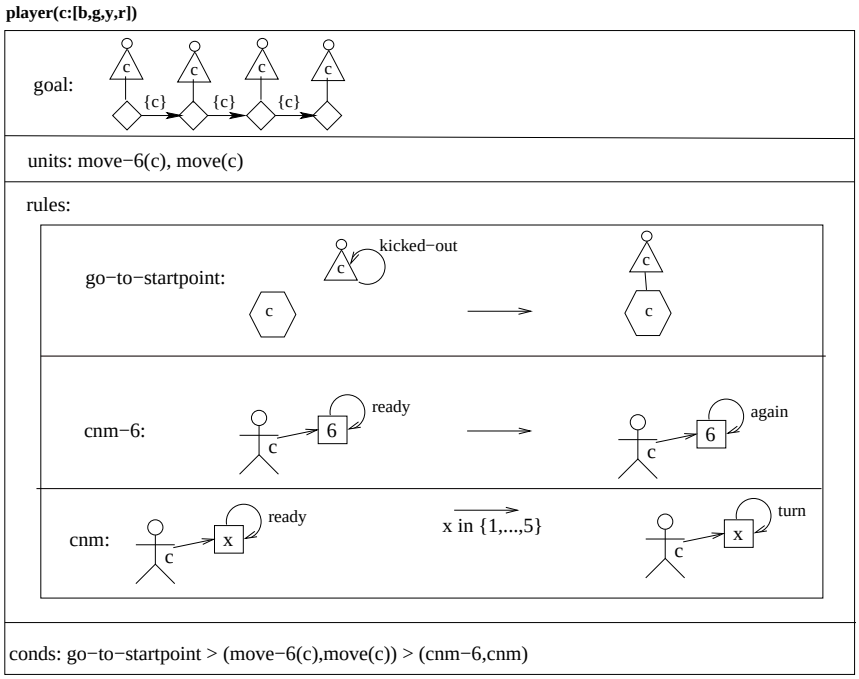


Figure 2. A Ludo player

should be always applied if possible. The rule *cnm-6* and the rules represented by *cnm* are applied if no token can be moved by the player, i.e. they have the lowest priority. The rule *cnm-6* asks the die to throw again and *cnm* asks the die to turn to the next player if a number between one and five was thrown.

Every player contains the two transformation units *move-6* and *move*. The transformation unit *move* is depicted in Fig. 3. It models all moves of a token if no six is thrown. (The moves corresponding to a six are contained in the transformation unit *move-6*. It is similar to *move* and therefore not depicted.) The transformation unit *move* contains four rules. The first, *mf* moves a token from the first round node (the one connected to its start node) $x$ nodes ahead where $x \in \{1, \ldots, 5\}$ is the number thrown by the die. This move can only be performed if the target node is not occupied and if there is still a token at the start node. Moreover, the token can only be moved if it is the turn of its player. This is indicated by the arrow pointing from player $c$ to the die. On the left-hand side the die has a *ready*-loop which means that the die has already thrown itself. On the right-hand side the die is asked to turn. The rule *mfko* is similar to *mf*. The difference is that another token is kicked out. The rule *go* moves a token from a round or a home node to another round or home node. The rule *goko* moves a token from a round node to another node where it kicks out a token. The rules *go* and *goko* can only be applied if the first two rules are not applicable, because the first round node must be left if it is occupied by a token of color $c$ and if there is still a token at the start node. If this move is not possible any other token can be taken.

It is worth noting that players select their tokens nondeterministically. More sophisticated rules would allow to decide whether it is appropriate to choose a token that can kick out another one, etc. The rules for making such decisions possible are more complicated, because they have to consider a wider context of the environment (e.g. such a rule could check whether the kicking out of another token brings

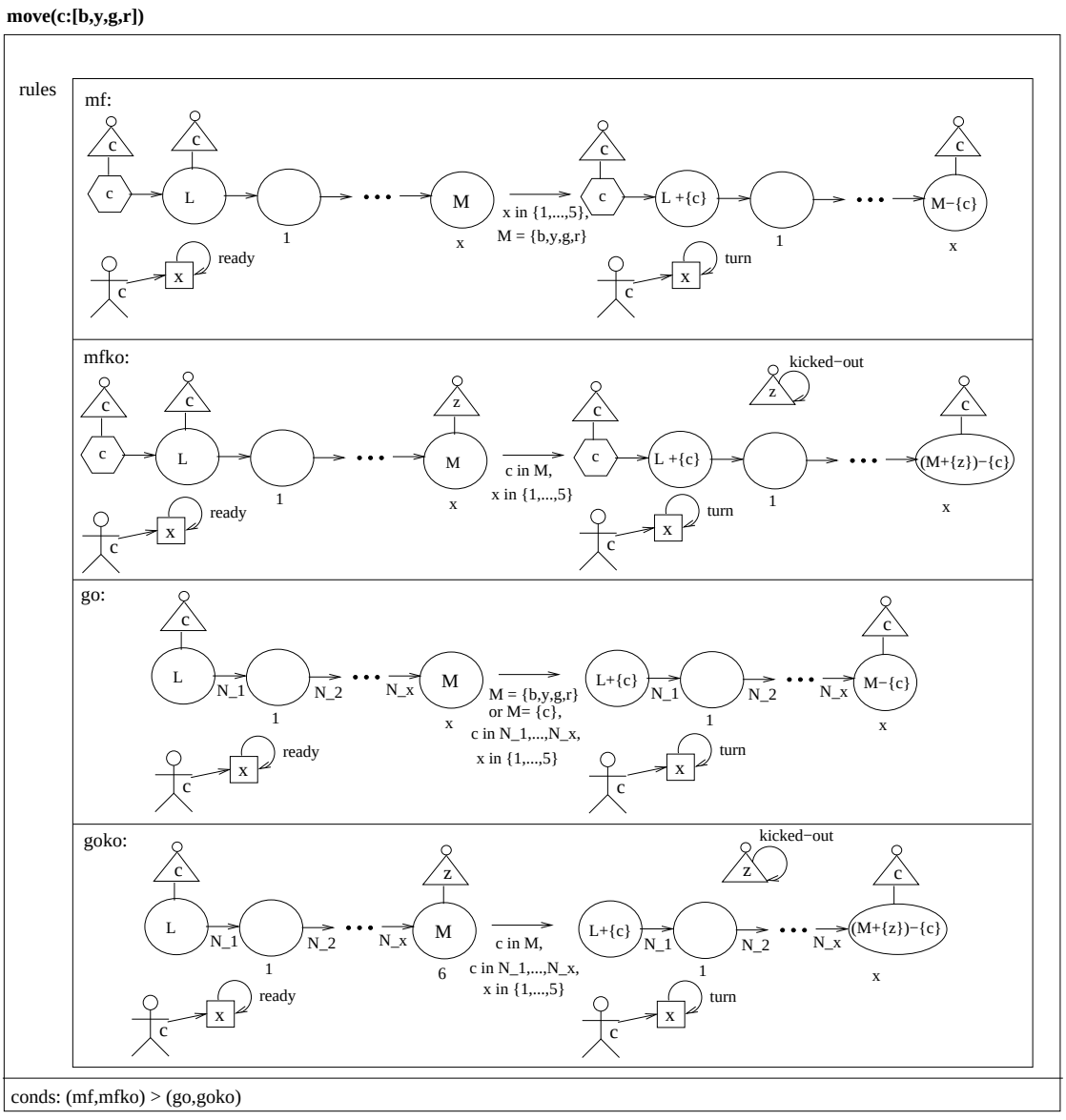


Figure 3. The transformation unit *move*

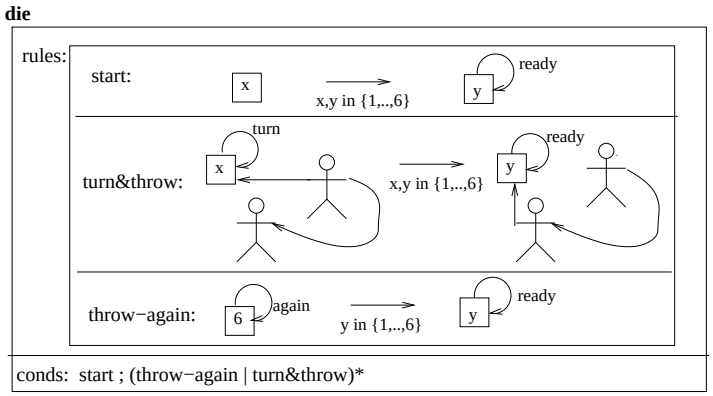


Figure 4. The autonomous unit *die*

the own token into a ”dangerous” position). For reasons of space limitations they are kept simple in this paper.

The other autonomous unit of *Ludo* models the die and is depicted in Fig. 4. The autonomous unit *die* has no special goal, i.e. it admits every graph as a goal. The only functionality of *die* is to throw itself and to move to the next player. The first rule throws the die. The second rule turns and throws the die in the case where the die gets a corresponding *turn*-message from the player. With the third rule the die throws itself again without moving to the next player. This can be only done if a six was thrown before. The control condition requires that the *start* rule be applied once at first. Afterwards any of the two remaining rules can be applied arbitrarily often.

The game *Ludo* (including the board, up to four players, one die and the game rules) can be modeled as the community *Ludo*$= (\mathit{one}(\mathit{player}), \mathit{Iniconf}, \{\mathit{player}(b), \mathit{player}(y), \mathit{player}(g), \mathit{player}(r), \mathit{die}\})$ where the overall goal $one(player)$ specifies all graphs in which at least one player unit has reached its goal and *Iniconf* specifies all possible initial environments of *Ludo*.

The $Ludo$ case study has been implemented and tested in Java. The implementation is based on the graph transformation engine of the AGG system (cf. [5]). Since this system neither provides all of the necessary control conditions nor supports transformation units, two attempts have been made. In the first attempt, the $Ludo$ community has been translated into a single flat AGG graph transformation grammar. Here the missing control conditions have been simulated by extensions of the existing rules. In a second attempt, the actual $Ludo$ community with its individual autonomous units have been implemented in Java. In this attempt, the generic implementation controls the derivation process, but the actual graph transformation is performed by AGG with graph transformation rules corresponding to the original specification presented here. The second version is performing significantly faster, especially due to the optimization not to test all the rules of all autonomous units but only those of the autonomous unit that actually has to move. Fig. 5 shows a screenshot of the environment graph in the running system. Here the blue player has just rolled a score of 1 and moved one of its tokens, kicking out a red token in the process. In Fig. 6 a different visualization is shown, which looks more similar to the actual board game. In the second version each player unit employs a different strategy. One autonomous unit kicks out other players' tokens whenever it has the chance to do so. One autonomous unit tries to play more defensively, i.e., if different tokens can be moved then it prefers to move without kicking out other players' tokens. A

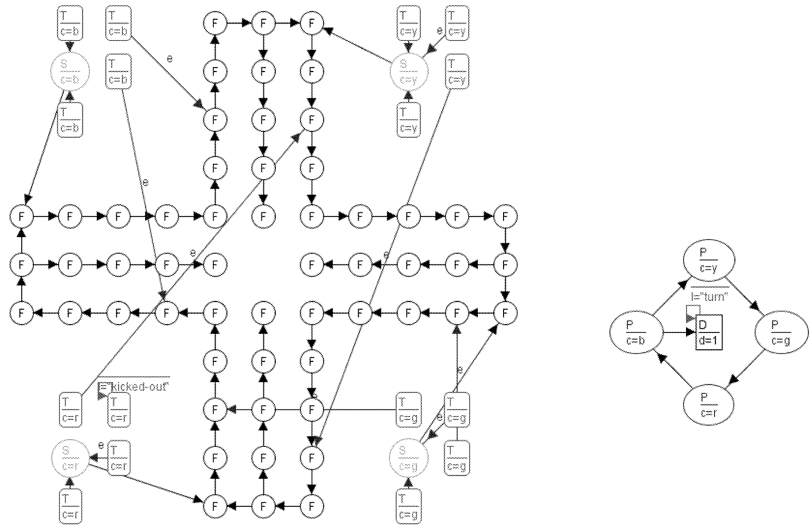


Figure 5. A screenshot of the implemented system

third strategy is trying to always move the token that has covered the most distance in order to reach the target as fast as possible. On the contrary, the fourth autonomous unit always tries to move the token that has covered the least distance in order to move in a compact way. These different strategies are specified in the corresponding player units using higher priorities for the rules according to the desired behavior.

# 6. Parallel-process semantics

In this section, we generalize the framework of autonomous units by permitting that autonomous units act and interact in parallel.

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.

**Definition 6.1. (Multisets)**

1. Given some basic domain $D$, the set of all multisets $D_*$ over $D$ with finite carriers consists of all mappings $m\colon D \to \mathbb{N}$ such that the carrier $car(m) = \{d \in D \mid m(d) \neq 0\}$ is finite.

2. For $d \in D, m(d)$ is called the multiplicity of $d$ in $m$.

**Remarks.**

1. The union or sum of multisets can be defined by adding corresponding multiplicities.

2. $D_*$ with this sum is the free commutative monoid over $D$ where the multiset with empty carrier is the null element, i.e. $null\colon D \to \mathbb{N}$ with $null(D) = 0$.

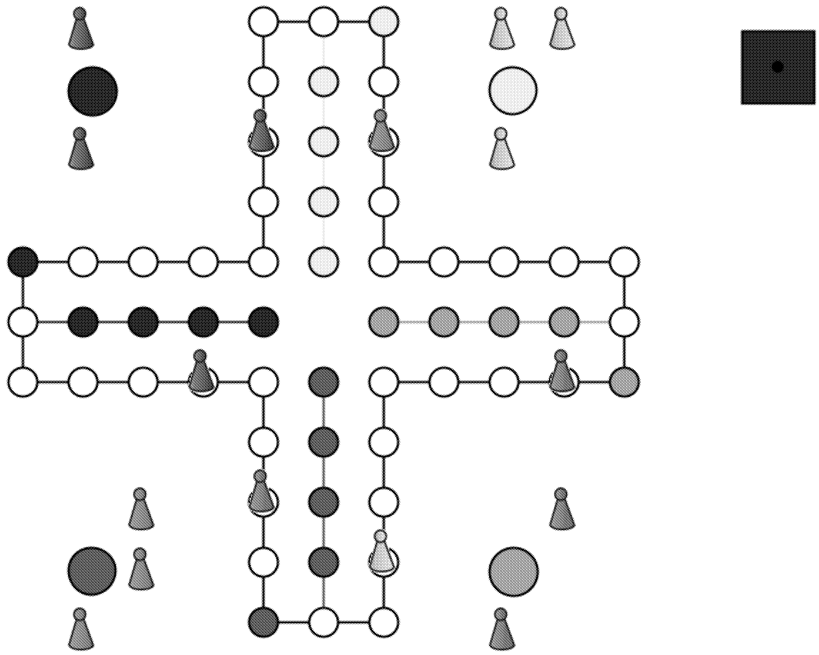

Figure 6. A screenshot of the implemented system with a different visualization

3. Note that the elements of $D$ correspond to singleton multisets, i.e. for $d \in D, \hat{d}\colon D \to \mathbb{N}$ with $\hat{d}(d) = 1$ and $\hat{d}(d') = 0$ for $d' \neq d$.

4. 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 yielding a graph models the parallel and multiple application of several rules.

The definitions of a graph transformation approach, an autonomous unit and a community of autonomous units remain unchanged with two exceptions.

1. We assume now that not only each rule, but each multiset of rules $r \in \mathcal{R}_*$ specifies a binary relation on graphs $SEM(r) \subseteq \mathcal{G} \times \mathcal{G}$. The multisets of rules in $\mathcal{R}_*$ are called parallel rules. A graph transformation approach with paralled rules is called a parallel graph transformation approach. The application of a parallel rule $r$ to $G$ with the result $G'$. may be also called a direct parallel derivation or a parallel derivation step.

2. We consider only autonomous units without transformation units to keep the technicalities simple. Hence, an autonomous unit $aut = (g, \emptyset, P, c)$ is denoted by $aut = (g, P, c)$.

**Example of a parallel graph transformation approach.** The sample approach introduced in Section 2 with directed graphs and dpo-rules is particularly suited to be extended into a parallel approach. 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 exist graphs $M_1$ and

$M_2$ such that $(G, M_1)$ and $(M_2, G')$ are in $SEM(r_1)$ and $(G, M_2)$ and $(M_1, G')$ are in $SEM(r_2)$. This means that the graph $G'$ can also be obtained from $G$ by applying the rules $r_1$ and $r_2$ sequentially and in any order. Moreover, let $r_i$ ($i = 1, 2$) be two (parallel) rules and let $g_i \colon L_i \to G$ be two morphisms that satisfy the gluing condition described in steps (1) and (2) of a rule application. Then $r_1$ and $r_2$ are independent w.r.t. $g_i$ if the the following independence condition is satisfied:

$$g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2).$$

In this case both rules can be applied to $G$ in parallel via the application of $r_1 + r_2$ using the graph morphism $\langle g_1, g_2 \rangle \colon L_1 + L_2 \to G$ such that $\langle g_1, g_2 \rangle(x) = g_i(x)$ if $x$ is an element of $L_i$ (see, e.g., [2] for more details).

As in the sequential case, an autonomous unit modifies an underlying environment in the parallel setting too while striving for its goal. Its semantics consists of a set of transformation processes being finite or infinite sequences of environment transformations. An environment transformation comprises the parallel application of local rules or environment changes performed by other autonomous units that are working in the same environment. Because the parallel-process semantics is meant to describe the simultaneous activities of autonomous units, the environment changes must be possible while a single autonomous unit applies its rules. To achieve this, we assume that there are some rules, called metarules, the application of which defines environment changes. Consequently, environment changes and ordinary rules can be applied in parallel. Hence, a parallel transformation process of an autonomous unit consists of a sequence of parallel rule applications which combine local rule applications with environment changes specified by other components so that the control condition is satisfied.

**Definition 6.2. (Parallel semantics)**
Let $aut = (g, P, c)$ be an autonomous unit and let $Change \subseteq \mathcal{G} \times \mathcal{G}$. Let $\mathcal{MR} \subseteq \mathcal{R}_*$ be a set of parallel rules, called metarules, such that $SEM(\mathcal{MR}) = \bigcup_{r \in \mathcal{MR}} SEM(r) = Change$. Let $s = (G_0, G_1, G_2, \cdots) \in SEQ(\mathcal{G})$. Then $s \in PAR_{Change}(aut)$ if

- for $i = 1, \cdots, |s| - 1$ if $s$ is finite and for $i \in \mathbb{N}^+$ if $s$ is infinite, $(G_{i-1}, G_i) \in SEM(r + r')$ for some $r \in P_*$ and $r' \in \mathcal{MR}$, and
- $s \in SEM_{Change}(c)$.

**Remarks.**

1. Successful parallel transformation processes are defined as in the sequential case.
2. The elements of $PAR_{Change}(aut)$ are sequences of applications of parallel rules which may be called the parallel processes of $aut$. Every single step of these processes applies a parallel rule of the form $r + r'$ where $r$ is a parallel rule of the unit $aut$ and $r'$ is a metarule. Therefore, while the autonomous unit acts on the environment graph, the environment may change in addition. But as $r$ and $r'$ may be the null rule and $r + null = r$ as well as $null + r' = r'$, a step can also be an exclusive activity of $aut$ or a change of the environment only.

The following definition generalizes the change relation of an autonomous unit in a community to the parallel case. In particular, it takes into account all parallel rule applications that can be performed by the other units in the community.

**Definition 6.3. (Change relation)**
Let $CAU = (Goal, Init, Aut)$ be a community. Then for each $aut \in Aut$ the *change relation* $Change(aut)$ w.r.t. $aut$ is given by the parallel rules composed of rules of the autonomous units in $CAU$ other than $aut$ as metarules, i.e. $Change(aut) = SEM(\bigcup_{aut' \in Aut - \{aut\}} (P_{aut'})_*)$.

Every transformation process of a community starts with an initial environment and is contained in the parallel semantics of every autonomous unit participating in the community.

**Definition 6.4. (Parallel community semantics)**
For $CAU = (Goal, Init, Aut)$, the *parallel community semantics* of $CAU$, denoted by $PAR(CAU)$, consists of all finite or infinite sequences $s = (G_0, G_1, \ldots) \in SEQ(\mathcal{G})$ such that
$G_0 \in SEM(Init)$ and $s \in PAR_{Change(aut)}(aut)$ for all $aut \in Aut$.

**Remarks.**

1. Again successful parallel transformation processes of communities are defined as in the sequential case.

2. The properties given in the first two remarks after Definition 4.3 hold also for parallel semantics of communities.

# 7. Petri nets

The area of Petri nets (see, e.g., [13, 1]) is established as one of the oldest, well-known, and best studied frameworks in which parallelism is precisely introduced and investigated. Hence it is meaningful to relate Petri nets with the parallel semantics of communities of autonomous units and to shed some light on the significance of the latter in this way. It turns out for instance that place/transition nets, which are the most frequently used variants of Petri nets, can be seen as a special case of communities of autonomous units where the transitions play the role of the autonomous units.

A place/transition system $S = (P, T, F, m_0)$ consists of a set $P$ of places, a set $T$ of transitions, a flow relation $F \subseteq (P \times T) \cup (T \times P)$, and an initial marking $m_0 : P \to \mathbb{N}$, i.e. $m_0 \in P_*$. The sets $P$ and $T$ are assumed to be disjoint so that $N = (P \cup T, F)$ is a bipartite graph (with the projections as source and target maps respectively).

The firing of enabled transitions transforms markings that are multisets of places. This is formally defined as follows.

A multiset $m \in P_*$ is called a marking. A transition $t \in T$ is enabled w.r.t. $m$ if ${}^\bullet t \leq m$ where ${}^\bullet t : P \to \mathbb{N}$ describes the input places of $t$ that flow into $t$, i.e. ${}^\bullet t(p) = 1$ if $(p, t) \in F$ and ${}^\bullet t(p) = 0$ otherwise. Moreover, ${}^\bullet t \leq m$ is defined place-wise, i.e. ${}^\bullet t(p) \leq m(p)$ for all $p \in P$ or, in other words, $m(p) \neq 0$ if $(p, t) \in F$. If $t$ is enabled w.r.t. $m$, it can fire resulting in a marking which is obtained by subtracting ${}^\bullet t$ from $m$ and by adding $t^\bullet$ given by $t^\bullet(p) = 1$ if $(t, p) \in F$ and $t^\bullet(p) = 0$ otherwise. Such

a firing is denoted by $m\,[t\rangle\, m - {}^\bullet t + t^\bullet$. If one interprets $m(p)$ as the number of tokens on the place $p$, then the firing of $t$ removes one token from each input place of $t$ and puts a new token on each of the output places of $t$.

Analogously, a multiset of transitions $\tau \in T_*$ can be fired in parallel by summing up all input places and all output places:

$$m\,[\tau\rangle\, m - {}^\bullet\tau + \tau^\bullet \text{ provided that } {}^\bullet\tau \leq m.$$

Here ${}^\bullet\tau$ and $\tau^\bullet$ are defined by ${}^\bullet\tau(p) = \sum_{t\in T} \tau(t) * {}^\bullet t(p)$ and $\tau^\bullet(p) = \sum_{t\in T} \tau(t) * t^\bullet(p)$ for all $p \in P$, and ${}^\bullet\tau \leq m$ is again place-wise defined, i.e. ${}^\bullet\tau(p) \leq m(p)$ for all $p \in P$.

Now one may consider the underlying net, which is the bipartite graph $N$, together with a marking as an environment. This is represented by the marking because the net is kept invariant. The transitions can be seen as rules and the firing of multisets of transitions as parallel rule application. As environment class expressions, we need single markings describing themselves as initial markings and the constant $all$ accepting all environments. The only control condition needed is the constant $free$ allowing a unit the free choice of rules. Then these components form a graph transformation approach, and a place/transition system $S = (P, T, F, m_0)$ can be translated into a community of autonomous units $CAU(S) = (all, m_0, \{aut(t) \mid t \in T\})$ with $aut(t) = (all, \{t\}, free)$.

A parallel process of $CAU(S)$ is a sequence of markings $m_0 m_1 \ldots$ such that for each two successive markings $m_i$ and $m_{i+1}$, there is a multiset $\tau_{i+1}$ of transitions that is enabled by $m_i$ and yields $m_{i+1}$ if fired. Therefore one gets a firing sequence $m_0\,[\tau_1\rangle\, m_1\,[\tau_2\rangle \ldots$. Conversely, given such a firing sequence, one may remove the firing symbols including the multisets of transitions and obtain a parallel process of $CAU(S)$ as parallel rule application coincides with firing of multisets of transitions. This proves that the community of autonomous units $CAU(S)$ mimics the place/transition system $S$ correctly. Figure 7 depicts the relation. The adapter transforms a firing sequence into a sequence of markings by removing the firing symbol (including the fired multisets of transitions) between each two successive markings.

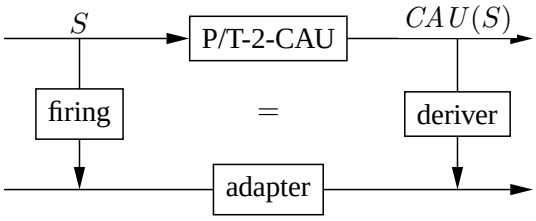


Figure 7. Correctness diagram for the translation of Petri nets into communities

## 8. Cellular automata

Cellular automata (see, e.g., [16]) are well-known computational devices that exhibit massive parallelism. A cellular automaton consists of a network of cells each in a particular state. In a computational step, all cells change their states in parallel depending on the states of their neighbors. To simplify technicalities, one may assume that the neighborhoods of all cells are regular meaning that they have the same number

of neighbors and that the state transition of all cells is based on the same finite-state automaton. This leads to the following formal definition.

A cellular automaton is a system $CA = (G, A, init)$ where

- $G = (V, E, s, t, l)$ is a regular graph of type $k$ subject to the condition: for each $v \in V$, there is a sequence of edges $e(v)_1 \cdots e(v)_k$ with $s(e(v)_i) = v$ and $l(e(v)_i) = i$ for all $i = 1, \ldots, k$,
- $A = (Q, Q^k, d)$ is a finite-state automaton, i.e. $Q$ is a finite set of states, $Q^k$ is the input set and $d \subseteq Q \times Q^k \times Q$ is the state transition with k-tuples of states as inputs, and
- $init\colon V \to Q$ is the initial configuration.

If the graph $G$ is infinite, one assumes a sleeping state $q_0 \in Q$ in addition such that $d(q_0, q_0^k) = \{q_0\}$ and $active(init) = \{v \in V \mid init(v) \neq q_0\}$ is finite.

The latter means that only a finite number of nodes is not sleeping initially and that the sleeping state can only wake up if not all inputs are sleeping. The edge sequence $e(v)_1 \cdots e(v)_k$ yields the neighbors of $v$ as targets, i.e. $t(e(v)_1) \cdots t(e(v)_k)$.

A configuration is a mapping $con\colon V \to Q$ that assigns each node (which represent cells) an actual state. Configurations can be updated by state transitions of all actual states using the states of the neighbors as input.

Let $con\colon V \to Q$ be a configuration. Then $con'\colon V \to Q$ is a directly derived configuration, denoted by $con \vdash\!\!\!- con'$, if the following holds for every $v \in V$:

$$con'(v) \in d(con(v), con(t(e(v)_1)) \cdots con(t(e(v)_k))).$$

The semantics of a cellular automaton $CA$ is given by all configurations that can be derived from the initial configuration:

$$L(CA) = \{con \mid init \vdash\!\!\!-^* con\}$$

It is worth noting and easy to prove that all configurations derivable from the initial configuration have a finite number of nodes with non-sleeping states. Typical examples of regular graphs underlying cellular automata are the following: The set of nodes is the set of all points in the plane with integer coordinates, i.e. $\mathbb{Z} \times \mathbb{Z}$. Then there are various choices for the neighborhood of a node $(x, y) \in \mathbb{Z} \times \mathbb{Z}$. that establish the set of edges with sources and targets. Typical ones are:

1. the four nearest nodes (to the north, east, south and west): $(x, y+1), (x+1, y), (x, y-1), (x-1, y)$,
2. the eight nearest nodes: $(x, y+1), (, x+1, y+1), (x+1, y), (x+1, y-1), (x, y-1), (x-1, y-1), (x-1, y), (x-1, x+1)$,
3. only the neighbors to the south and the west: $(x, y-1), (x-1, y)$.

The edges connecting a node with a neighbor may be numbered in the given order.

Cellular automata can be translated into communities of autonomous units where each cell is transformed into an autonomous unit.

The environments are given by the configurations. To get a graph representation of a configuration $con$, the underlying regular graph $G$ is extended by a loop at each node $v$ which is labeled with $con(v)$,

i.e. $(G, con) = (V, E + V, \overline{s}, \overline{t}, \overline{l})$, such that $G$ is a subgraph and $\overline{s}(v) = \overline{t}(v) = v$ and $\overline{l}(v) = con(v)$ for all $v \in V \subseteq E + V$.

The community $CAU(CA)$ associated with a cellular automaton $CA = (G, A, init)$ gets $(G, init)$ as initial environment and an autonomous unit $aut(v)$ for each $v \in V$.

Each of these units has the same rules with positive context which reflect the state transitions. They are represented by the rule in Fig. 8 and can be applied if $q' \in d(q, q_1, \cdots, q_k)$ and not all the states $q, q_1, \ldots, q_k$ are sleeping.

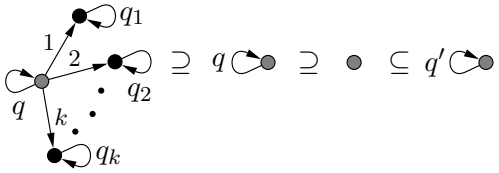


Figure 8. Graph transformation rule with positive context modeling the transitions of a cellular automaton

Moreover each unit $aut(v)$ has got a control condition requiring that the central node must be mapped to $v$. This means that the matching of the left-hand side of each rule is fixed and no search for it is needed. Moreover, the matchings of rules of different units are not overlapping so that the rules can be applied in parallel. If a node is sleeping and all its neighbors are sleeping too, then no rule can be applied. A parallel rule is maximal if all other nodes are matched. According to this construction, the application of such a maximal parallel rule to the environment $(G, con)$ yields an environment $(G, con')$ such that $con \vdash\!\!\!\!- con'$. This means that the application of a maximal parallel rule corresponds exactly to a derivation step on the respective configurations.

To put it in another way, the semantics of a cellular automaton $CA$ and the parallel semantics $PAR(CAU(CA))$ of the community of autonomous units $CAU(CA)$ are nicely related to each other if one applies maximal parallel rules only. Let $L(PAR(CAU(CA)))$ be the set of configurations $con$ such that a parallel process $(G, init) \cdots (G, con) \in PAR(CAU(CA))$ exists. Then $L(PAR(CAU(CA)))$ equals $L(CA)$. This correctness result is depicted in Fig. 9.

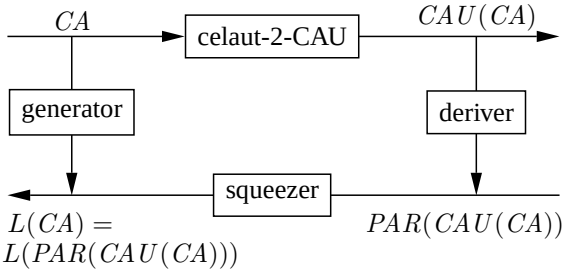


Figure 9. Correctness diagram for the translation of cellular automata into communities

A finite-state automaton fitting the third neighborhood is $SIER = (\{b, w\}, \{b, w\}^2, d)$ where the state transition is defined as $d(b, x, y) = b$ for all $x, y \in \{b, w\}, d(w, b, w) = d(w, w, b) = b$, and $d(w, b, b) = d(w, w, w) = w$. The state $w$ is sleeping.

The initial configuration may map the node $(0, 0)$ to $b$ and all others to $w$.

There is a very nice pictorial interpretation of this cellular automaton. Each node $(x, y)$ is represented by the square spanned by the points $(x, y), (x, y + 1), (x + 1, y + 1), (x + 1, y)$. If a configuration $con$ assigns $b$ to $(x, y)$, the square gets the color black and white otherwise. The initial configuration consists of a single black square. Because the automaton is deterministic, there is exactly one derivation for each length, where the shorter derivations are initial sections of the longer ones. The first five steps are depicted in Fig. 10.

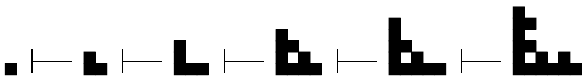

Figure 10. Visualization of a transformation process

After 15 steps the picture looks as shown in Fig. 11. All derived configurations can be seen as approximations of the Sierpinski triangle, a famous fractal. (see, e.g., [12]).

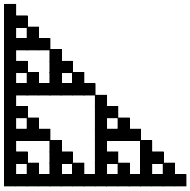

Figure 11. Visualized environment after 15 environment changes

## 9. Multiagent systems

Multiagent systems are modeling and programming devices well-known in artificial intelligence (see, e.g., Wooldridge et al. [17]). A multiagent system provides a set of agents and an initial environment state. Starting at this state, the agents change environment states step by step where they act together in parallel in each step. Each agent can perceive the current environment state at least partly. Based on this perception and its own intention, the agent chooses an action to be performed next. Therefore, a process in a multiagent system $MAS$ is a sequence

$$es_0\ es_1\ es_2\ \cdots$$

of environment states $es_i$ for all $i$ where $es_0$ is initial. Each environment state $es_{i+1}$ is given by the state transition $\tau$ of $MAS$ depending on the previous state $es_i$ and the action $act(ag)_i$ chosen by every agent $ag$ of $MAS$. The choice of such an action is made according to the function $do_{ag}$ each agent $ag$ is provided with. The $do$-function yields an action depending on the agent's perception $perceive_{ag}(es_i)$ of the current state and the agent's intention $intend_{ag}$. The global state transition $\tau$ and the functions $do_{ag}, perceive_{ag}$ and $intend_{ag}$ which are individually assigned to each agent $ag$ of $MAS$ are assumed to satisfy some consistency properties (cf. [17] for details). Altogether, multiagent systems form a logical

and axiomatic approach to model distributed information processes that interact on common environment states. It should be noted that all functions of $MAS$ are allowed to be nondeterministic so that chosen actions as well as the next state may not be uniquely determined.

Communities of autonomous units are nicely related to multiagent systems as may be not too surprising from the description above. Actually, a community of autonomous units $CAU = (Goal, Init, Aut)$ turns out to be a particular rule-based model of multiagent systems. The environment states are the environment graphs. The agents are the autonomous units. The initial graphs are explicitly given. The rules – or the parallel rules likewise – of a unit are the actions of the agent embodied by the unit. The control condition plays the role of the $do$-function because it identifies the rules that are allowed to be applied next. As the control condition can take into account the current environment graph, the perception of the agent is also reflected. The most important aspect of the correspondence between agents and autonomous units is the transition function that is made operational by means of parallel rule application. The parallel rule to be applied in each step is just the sum of all rules chosen by the various autonomous units according to their control. If one considers the parallel rules of a unit as actions, the parallel processes of the community and the processes of the corresponding multiagent system coincide. If only the rules are actions, the multiagent system is not parallel with respect to single agents. That all agents must act in parallel in each step is a minor difference to community processes because a multiagent system may provide void actions without effect to the environment.

The relation between communities of autonomous units and multiagent systems is only sketched because a full formal treatment is beyond the scope of the paper. But even on this informal level, it should be clear that both concepts fit nicely together and may profit from each other. Communities of autonomous units represent explicit models of multiagent systems on one abstract, implementation-independent level with a precise, rule-based operational semantics. The *perceive-do* mechanism of multiagent systems to choose next actions provides a wealthy supply of control conditions that can be employed in modeling by means of autonomous units.

## 10. Conclusion

In this paper, we have introduced communities of autonomous units as a means for modeling systems in which different components interact in a rule-based, self-controlled, and goal-driven manner within a common environment. Communities of autonomous units have been provided with a formal operational semantics based on interacting sequential and parallel processes. We have illustrated the notion of communities with a case study modeling the board game *Ludo* in which every player as well as the die can act as an autonomous unit. Moreover, we have studied the relationship of autonomous units to three other modeling frameworks that provide notions of parallelism: Petri nets, cellular automata, and multiagent systems. While the first two have been correctly transformed into autonomous units, autonomous units have turned out to be models of multiagent systems in that the environments are instantiated as graphs, the actions of agents as rules, and the environment transformation as parallel rule application.

The underlying formal framework for communities of autonomous units has been graph transformation which is highly adequate if the common environment can be represented in a natural way as a graph as for example in the case of board games and logistic applications. Nevertheless, it is worth noting that the graphs and the graph transformation rules the autonomous units are working with are not further specified in the underlying graph transformation approach so that in general, one can take as formal ba-

sis any rule-based mechanism that provides a set of configurations and a set of rules specifying a binary relation on such configurations.

There are at least the following interesting points for future work.

- The basic idea of autonomous units is that each of them decides for itself which rule is to be applied next. They are independent of each other and the parts of the environment graphs where their rules apply may be far away from each other. Hence, a sequential behavior of the community (like in many card and board games) will be rarely adequate. But also the parallel behavior does not always reflect the actual situations to be modeled because a parallel step has an explicit begin and end whereas there may be activities of units that cannot be related to each other with respect to time. A proper concurrent semantics of autonomous units may fix this problem.
- Besides Petri nets, the theory of concurrency offers a wide spectrum of notions of processes like communicating sequential processes, calculus of communicating systems, traces, and bigraphs. A detailed comparison of them with autonomous units can lead to interesting insights.
- Communities of autonomous units should be implemented in order to be able to elaborate and to verify case-studies of realistic size. Currently there is being done some work in this direction at the University of Bremen which has as one aim to allow to plug in other already existing graph transformation tools as described in [4].
- Up to now, the goal of an autonomous unit is defined as a graph class expression. Since for some applications this may not be sufficient, other adequate classes of goals should be studied.

## Acknowledgement

We are grateful to the anonymous referees for their valuable comments.

## References

[1] Bause, F., Kritzinger, P.: *Stochastic Petri Nets - An Introduction to the Theory (Second Edition)*, Vieweg & Sohn, 2002.

[2] 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 [14], 163–245.

[3] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., Eds.: *Fundamentals of Algebraic Graph Transformation*, Springer, 2006.

[4] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G., Eds.: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*, World Scientific, 1999.

[5] Ermel, C., Rudolf, M., Taentzer, G.: The AGG-Approach: Language and Environment, in: Ehrig et al. [4], 551–603.

[6] Hölscher, K., Klempien-Hinrichs, R., Knirsch, P., Kreowski, H.-J., Kuske, S.: Autonomous Units: Basic Concepts and Semantic Foundation, *Understanding Autonomous Cooperation and Control in Logistics The Impact on Management, Information and Communication and Material Flow* (M. Hülsmann, K. Windt, Eds.), Springer, 2007.

[7] Hölscher, K., Kreowski, H.-J., Kuske, S.: Autonomous Units and their Semantics — the Sequential Case, *Proc. 3rd Intl. Conference on Graph Transformations (ICGT 2006)* (A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg, Eds.), 4178, Springer, 2006.

[8] Kennedy, J., Eberhart, R. C.: *Swarm Intelligence*, Morgan Kaufmann, 2001.

[9] Kreowski, H.-J., Kuske, S.: Graph Transformation Units with Interleaving Semantics, *Formal Aspects of Computing*, **11**(6), 1999, 690–723.

[10] Kreowski, H.-J., Kuske, S.: Autonomous Units and Their Semantics - The Parallel Case, *Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006* (J. Fiadeiro, P. Schobbens, Eds.), 4408, 2007.

[11] Kuhn, A.: Prozessketten – Ein Modell für die Logistik, in: *Erfolgsfaktor Logistikqualität* (H. Wiesendahl, Ed.), Springer, 2002, 58–72.

[12] Peitgen, H., Jürgens, H., Saupe, D.: *Chaos and Fractals*, Springer, 2004.

[13] Reisig, W.: *Elements of Distributed Algorithms: Modeling and Analysis With Petri Nets*, Springer Verlag, 1998.

[14] Rozenberg, G., Ed.: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, World Scientific, 1997.

[15] Scheer, A.: *Vom Geschäftsprozeß zum Anwendungssystem*, Springer, 2002.

[16] Wolfram, S.: *A New Kind of Science*, Wolfram Media, Inc., 2002.

[17] Wooldridge, M., Jennings, N. R.: Intelligent Agents: Theory and Practice, *The Knowledge Engineering Review*, **10**(2), 1995.