

# Graph Grammars and Petri Net Transformations

Hartmut Ehrig and Julia Padberg

Technical University Berlin, Germany  
Institute for Software Technology and Theoretical Computer Science  
{ehrig,padberg}@cs.tu-berlin.de

**Abstract.** The aim of this paper is a tutorial introduction to graph grammars and graph transformations on one hand and to Petri net transformations on the other hand. In addition to an introduction to both areas the paper shows how they have influenced each other. The concurrency concepts and semantics of graph transformations have been generalized from those of Petri net using the fact that the token game of Petri nets can be considered as a graph transformation step on discrete graphs. On the other hand each Petri net can be considered as a graph, such that graph transformations can be used to change the net structure of Petri nets. This leads to a rule based approach for the development of Petri nets, where the nets in different development stages are related by Petri net transformations.

## 1 Introduction

The main idea of graph grammars is the rule-based modification of graphs where each application of a graph rule leads to a graph transformation step. Graph grammars can be used on one hand to generate graph languages in analogy to the idea to generate string languages by Chomsky grammars in formal language theory. On the other hand graphs can be used to model the states of all kinds of systems which allows to use graph transformations to model state changes of these systems. This allows to apply graph grammars and graph transformation systems to a wide range of fields in computer science and other areas of science and engineering. A detailed presentation of different graph grammar approaches and application areas of graph transformations is given in the 3 volumes of the *Handbook of Graph Grammars and Computing by Graph Transformation* [32, 10, 15].

The intention of the first part of this paper is to give a tutorial introduction to the basic concepts and results of one specific graph transformation approach, called double-pushout approach, which is based on pushout constructions in the category of graphs and graph morphisms. Although this approach is based on a categorical concept, we do not require that the reader is familiar with category theory: In fact, we introduce the concept of a pushout in the category of graphs from an intuitive point of view, where a pushout of graphs corresponds to the gluing of two graphs via a shared subgraph.

In Section 2 of this paper we give a general overview of graph grammars and graph transformations including the main approaches considered in literature.

The basic concepts of the double-pushout approach are introduced in Section 3 using as example the Pacman game considered as a graph grammar. Concepts and results concerning parallel and sequential independence as well as parallelism of graph transformations are introduced in Section 4. The main results are the local Church-Rosser Theorem and the Parallelism Theorem. The relationship between graph grammars and Petri nets is discussed in Section 5 of this paper. First we show how the basic concepts of both areas correspond to each other. Then we give an overview of the concurrent semantics of graph transformations, which has been developed in analogy to the corresponding theory of Petri nets.

In the second part of this paper we give an introduction to concepts and results of Petri net transformations. This area of Petri nets has been introduced about 10 years ago in order to allow in addition to the token game of Petri nets, where the net structure of fix, also the change of the nets structure [31, 28] This allows the stepwise development of Petri nets using a rule-based approach in the sense of graph transformations, where the net structure of a Petri net is considered as a graph. An intuitive introduction to Petri net transformations is given in Section 6 using the stepwise development of Petri nets for a baggage handling system as an example.

In Section 7 we show how the basic concepts of graph transformation - introduced in Section 3 for the double-pushout approach - can be extended to Petri net transformations in the case of place/transition nets. In addition we discuss a general result concerning the compatibility of horizontal structuring and transformation of Petri nets, which has been used in the example of Section 6. Moreover we give an overview of results as well for other Petri net classes, which kind of Petri net transformations are preserving interesting properties like safety and liveness.

The conclusion in Section 8 summarizes the main ideas of this paper and further aspects concerning graph grammars and Petri net transformations.

## 2 General Overview of Graph Grammars and Graph Transformation

The research area of graph grammars or graph transformations is a discipline of computer science which dates back to the early seventies. Methods, techniques, and results from the area of graph transformations have already been studied and applied in many fields of computer science such as formal language theory, pattern recognition and generation, compiler construction, software engineering, concurrent and distributed systems modeling, database design and theory, logical and functional programming, AI, visual modeling, etc.

The wide applicability is due to the fact that graphs are a very natural way of explaining complex situations on an intuitive level. Hence, they are used in computer science almost everywhere, e.g. as data and control flow diagrams, entity relationship and UML diagrams, Petri nets, visualization of software and hardware architectures, evolution diagrams of nondeterministic processes, SADT diagrams, and many more. Like the *token game* for Petri nets, a graph transfor-

mation brings dynamic to all these descriptions, since it can describe the evolution of graphical structures. Therefore, graph transformations become attractive as a *modeling and programming paradigm* for complex-structured software and graphical interfaces. In particular, graph rewriting is promising as a comprehensive framework in which the transformation of all these very different structures can be modeled and studied in a uniform way.

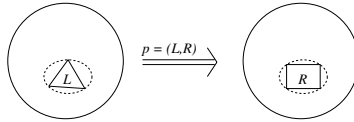
Before we go into more detail let us discuss the basic question

## 2.1 What Is Graph Transformation?

In fact, graph transformation has at least three different roots

- from Chomsky grammars on strings to graph grammars
- from term rewriting to graph rewriting
- from textual description to visual modeling.

Altogether we use the notion graph transformation to comprise the concepts of graph grammars and graph rewriting. In any case, the main idea of graph transformation is the rule-based modification of graphs as shown in Figure 1.



**Fig. 1.** Rule-based Modification of Graphs

The core of a rule or production  $p = (L, R)$  is a pair of graphs  $(L, R)$ , called left hand side  $L$  and right hand side  $R$ . Applying the rule  $p = (L, R)$  means to find a match of  $L$  in the source graph and to replace  $L$  by  $R$  leading to the target graph of the graph transformation. The main technical problem is how to connect  $R$  with the context in the target graph. In fact, there are different solutions how to handle this problem leading to different graph transformation approaches, which are summarized below.

## 2.2 Overview of Different Approaches

The main graph grammar and graph transformation approaches developed in literature so far are presented in the *Handbook of Graph Grammars and Computing by Graph Transformation vol 1: Foundations* [32].

1. The *node label replacement approach*, mainly developed by Rozenberg, Engelfriet and Janssens, allows replacing a single node as left hand side  $L$  by an arbitrary graph  $R$ . The connection of  $R$  with the context is determined by embedding rules depending on node labels.

2. The *hyperedge replacement approach*, mainly developed by Habel, Kreowski and Drewes, has as left hand side  $L$  a labeled hyperedge, which is replaced by an arbitrary hypergraph  $R$  with designated attachment nodes corresponding to the nodes of  $L$ . The gluing of  $R$  with the context at corresponding attachment nodes leads to the target graph.
3. The *algebraic approaches* are based on pushout and pullback constructions in the category of graphs, where pushouts are used to model the gluing of graphs. The double pushout approach, mainly developed by Ehrig, Schneider and the Berlin- and Pisa-groups, is introduced in Sections 3-5 in more detail.
4. The *logical approach*, mainly developed by Courcelle and Bouderon, allows expressing graph transformation and graph properties in modanic second order logic.
5. The *theory of 2-structures* was initiated by Rozenberg and Ehrenfeucht as a framework for decomposition and transformation of graphs.
6. The *programmed graph replacement approach* by Schuerr used programs in order to control the nondeterministic choice of rule applications.

### 2.3 Aims and Paradigms for Graph Transformation

Computing was originally done on the level of the *von Neumann Machine* which is based on machine instructions and registers. This kind of low level computing was considerably improved by assembler and high level imperative languages. From the conceptual - but not yet from the efficiency point of view - these languages were further improved by functional and logical programming languages. This newer kind of computing is mainly based on term rewriting, which - in the terminology of graphs and graph transformations - can be considered as a concept of tree transformations. Trees, however, do not allow sharing of common substructures, which is one of the main reasons for efficiency problems concerning functional and logical programs. This leads to consider graphs rather than trees as the fundamental structure of computing.

The main idea is to advocate graph transformations for the whole range of computing. Our concept of *Computing by Graph Transformations* is not limited to programming but includes also specification and implementation by graph transformations as well as graph algorithms and computational models and computer architectures for graph transformations.

This concept of Computing by Graph Transformations has been developed as basic paradigm in the ESPRIT Basic Research Actions COMPUGRAPH and APPLIGRAPH as well as in the TMR Network GETGRATS in the years 1990-2002. It can be summarized in the following way:

Computing by graph transformation is a fundamental concept for

- programming
- specification
- concurrency
- distribution
- visual modeling.

The aspect to support visual modeling by graph transformation is one of the main intentions of the ESPRIT TMR Network SEGRAVIS (2002-2006). In fact, there is a wide range of applications to support visual modeling techniques, especially in the context of UML, by graph transformation techniques. A state of the art report for applications, languages and tools for graph transformation on one hand and for concurrency, parallelism and distribution on the other hand is given in volumes 2 and 3 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [10] and [15]

### 3 Introduction to the DPO-Approach

As mentioned already in the general overview there are several algebraic graph transformation approaches based on pushout and pullback constructions in the category of graphs. The most prominent one is the double-pushout approach, short DPO-approach, initiated by Ehrig, Pfender and Schneider in [17]. The main idea is to model graph transformation by two gluing constructions for graphs and each gluing construction by a pushout. Roughly spoken, a production is given by  $p = (L, K, R)$ , where  $L$  and  $R$  are the left and right hand side graphs and  $K$  is a common interface of  $L$  and  $R$ . Given a production  $p = (L, K, R)$  and a context graph  $D$ , which includes also the interface  $K$ , the source graph  $G$  of a graph transformation  $G \Rightarrow H$  via  $p$  is given by the gluing of  $L$  and  $D$  via  $K$ , written  $G = L +_K D$ , and the target graph  $H$  by the gluing of  $R$  and  $D$  via  $K$ , written  $H = R +_K D$ . More precisely we will use graph morphisms  $K \rightarrow L$ ,  $K \rightarrow R$  and  $K \rightarrow D$  to express how  $K$  is included in  $L$ ,  $R$ , and  $D$  respectively. This allows to define the gluing constructions  $G = L +_K D$  and  $H = R +_K D$  as pushout constructions (1) and (2) leading to a double pushout in Figure 2.

$$\begin{array}{ccccc}
 L & \xleftarrow{\quad} & K & \xrightarrow{\quad} & R \\
 \downarrow & & \downarrow & & \downarrow \\
 & (1) & & (2) & \\
 G & \xleftarrow{\quad} & D & \xrightarrow{\quad} & H
 \end{array}$$

**Fig. 2.** DPO-Graph Transformation

Before we present more technical details of the DPO-approach, let us point out that it is based on graphs and total graph morphisms. In fact there is a slightly more general approach using graphs and partial graph morphism, where a graph transformation can be expressed by a single pushout. This approach has been initiated by Raoult and fully worked out by Lwe leading to the single pushout approach, short SPO-approach. A detailed presentation and comparison of both approaches is given in volume 1 of the handbook [32]. The DPO-approach has been generalized from graphs to any other kind of high-level structures. This leads to the theory of high-level replacement systems initiated in [12], which can

be applied to Petri nets leading to net transformation systems considered in Section 6 and Section 7 of this paper.

### 3.1 Graphs and Graph Morphisms

A *directed, labeled graph*  $G$ , short graph, over fixed sets of colors  $\Omega_E$  and  $\Omega_V$  for edges and vertices is given by

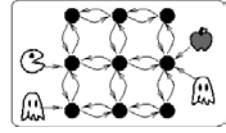
$$G = \Omega_E \xleftarrow{l_e} E \xrightleftharpoons[t]{s} V \xrightarrow{l_v} \Omega_V$$

**Fig. 3.** Directed Labeled Graph  $G$

where  $E$  and  $V$  are the sets of edges and vertices of  $G$ ,  $s$  and  $t$  are the source and target functions, and  $l_e$  and  $l_v$  are the edge and vertex label functions respectively.

An example for such a graph  $G$  is the Pacman graph  $PG$  in Figure 4, where the color  $*$  for the edges is omitted in  $PG$ .

- $\Omega_V = \{ \bullet, \odot, \ominus, \omin� \}$
- $\Omega_E = \{ * \}$
- Identities of nodes and edges are not shown explicitly



**Fig. 4.** Pacman Graph  $PG$  and Color Sets

A *graph morphism*  $f : G \rightarrow G'$  consists of a pair of functions  $f = (f_E : E \rightarrow E', f_V : V \rightarrow V')$ , which is compatible with source, target, and label functions of  $G$  and  $G'$ , i.e.  $f_V \cdot s = s' \cdot f_E$ ,  $f_V \cdot t = t' \cdot f_E$ ,  $l'_e \cdot f_E = l_e$  and  $l'_v \cdot f_V = l_v$ .

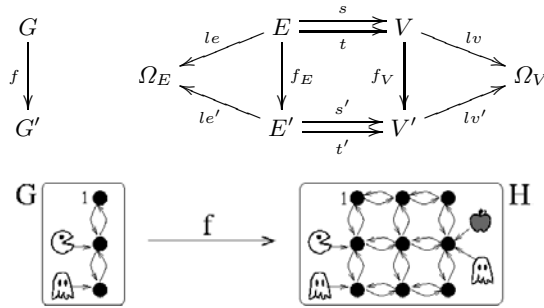
The diagram schema for graph morphisms and an example for a graph morphism is given in Figure 5.

The category **Graph** has graphs as objects and graph morphisms as morphisms.

Let us point out that there are also several other notions of graphs and graph morphisms which are suitable for the DPO-approach of graph transformation; especially typed graphs and attributed graphs, where the color sets are replaced by a type graph and a type algebra respectively.

### 3.2 Graph Productions and Graph Grammars

A *graph production*  $p = L \xleftarrow{l} K \xrightarrow{r} R$  consists of graphs  $L, K$  and  $R$  and (injective) graph morphisms  $l : K \rightarrow L$  and  $r : K \rightarrow R$  mapping the interface graph  $K$  to the left hand side  $L$  and the right hand side  $R$  respectively.

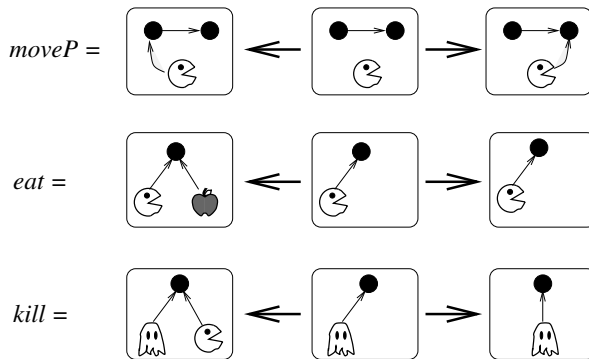


**Fig. 5.** Graph Morphism  $f : G \rightarrow G'$

A *graph grammar*  $GG = \{S, P, \Omega\}$  consists of a start graph, a set  $P$  of graph productions as defined above, and a pair of color sets  $\Omega = (\Omega_E, \Omega_V)$ , where  $S$  and the graphs in  $P$  are labeled over  $\Omega$ . An example is the Pacman graph grammar

$$PGG = \{PG, \{moveP, moveG, kill, eat\}\},$$

where the start graph  $PG$  is given in Figure 4 and the graph productions  $moveP$ ,  $moveG$ ,  $kill$ ,  $eat$  in Figure 6. The production  $moveG$  is similar to  $moveP$ , where pacman is replaced by the ghost. These productions allow pacman resp. the ghost to move along an arc of the grid of the pacman graph  $PG$ . The productions  $eat$  resp.  $kill$  allow pacman to eat an apple resp. the ghost to kill pacman, provided that pacman and the apple resp. ghost are on the same node of the grid.

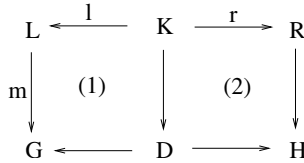


**Fig. 6.** Graph Productions of the Pacman Graph Grammar

Similar to Chomsky grammars it is also possible to distinguish between terminal and nonterminal color sets. In our case we have only terminal color sets. A graph grammar without distinguished start graph is also called *graph transformation system*.

### 3.3 Graph Transformation, Derivation and Graph Language

Given a graph production  $p = L \xleftarrow{l} K \xrightarrow{r} R$ , a graph  $G$  and a graph morphism  $m : L \rightarrow G$ , called *match* of  $L$  in  $G$ , then there is a *graph transformation*, also called *direct derivation*, if a double-pushout diagram as shown in Figure 7 can be constructed, where (1) and (2) are pushouts in the category **Graph**.



**Fig. 7.** Graph Transformation with Pushouts (1) and (2)

A *graph transformation* as given in Figure 7 is denoted by  $G \xRightarrow{p,m} H$ , or  $G \Rightarrow H$  via  $(p,m)$ , where  $G$  is the source graph and  $H$  the target graph. In the next section we will show that pushouts can be interpreted as gluing constructions. Given a production and a match  $m : L \rightarrow G$  means that we require to be able to construct a context graph  $D$  such that  $G$  is the gluing of  $L$  and  $D$  along  $K$  in pushout (1) and  $H$  is the gluing of  $R$  and  $D$  along  $K$  in pushout (2) of Figure 7, written

$$G = L +_K D \quad \text{and} \quad H = R +_K D.$$

The morphism  $R \rightarrow H$  in Figure 7 is called *comatch* of the graph transformation. A *graph transformation sequence*, also called *derivation*, is given by a finite sequence of graph transformations

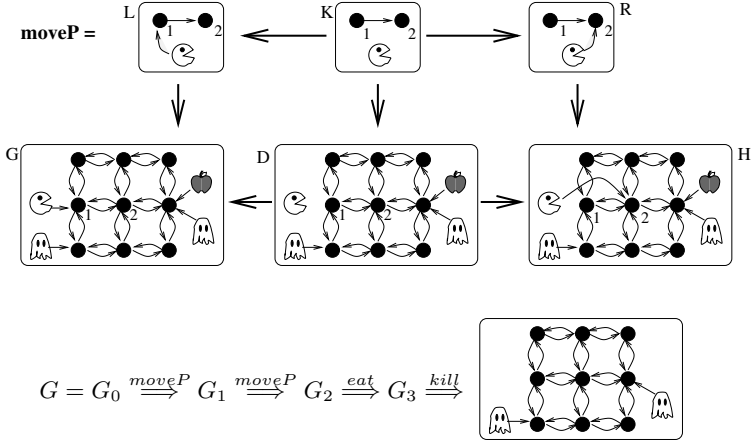
$$G_0 \xRightarrow{p_1, m_1} G_1 \xRightarrow{p_2, m_2} \dots \xRightarrow{p_n, m_n} G_n.$$

The *graph language* generated by a graph grammar  $GG = \{S, P, \Omega\}$  is the set of all graphs derivable from the start graph  $S$  with productions in  $P$ .

An example of a graph transformation using the production *moveP* is given in Figure 8, where pacman is moving from node 1 in graph  $G$  to node 2 in graph  $H = G_1$ . Moreover, Figure 8 shows a graph transformation sequence, where after this first step the productions *moveP* again, and also *eat* and *kill* are applied.

In Figure 8 it is intuitively clear that  $G$  is the gluing of  $L$  and  $D$  along  $K$  and  $H$  the gluing of  $R$  and  $D$  along  $K$ . Vice versa, given the production *moveP* and the match  $m : L \rightarrow G$  the context graph  $D$  can be constructed by removing from  $G$  all items of  $L$ , which are not in the interface  $K$ . In our case it is only the arc from pacman to node 1, which has to be removed. This is the first step in the explicit construction of a graph transformation. It leads to a pushout (1) in Figure 7 if a suitable *gluing condition* is satisfied which will be explained below. The second step is the gluing of  $R$  and  $D$  along  $K$  leading to pushout (2) in Figure 7.





**Fig. 8.** A Sample Graph Transformation and Derivation

In general the construction of a graph transformation  $G \xRightarrow{p,m} H$  from a production  $p = L \xleftarrow{l} K \xrightarrow{r} R$  and a match  $m : L \rightarrow G$  is given in two steps, where the first step requires that the gluing condition (see 3.5 below) is satisfied:

**STEP 1 (DELETE):** Delete  $m(L - l(K))$  from  $G$  leading to a context graph  $D$  (if the gluing condition is satisfied), s.t.  $G$  is the gluing of  $L$  and  $D$  along  $K$ , i.e.  $G = L +_K D$  in (1) of Figure 7.

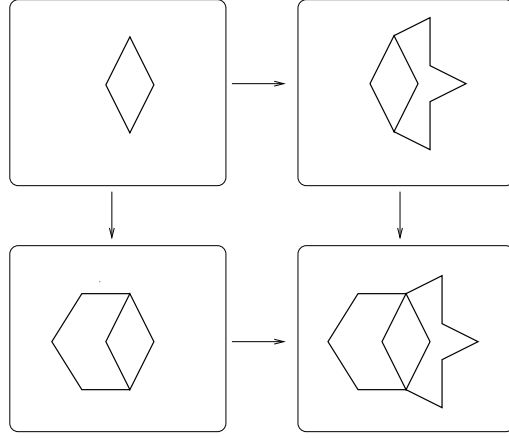
**STEP 2 (ADD):** Add  $R - r(K)$  to  $D$  leading to a graph  $H$ , s.t.  $H$  is the gluing of  $R$  and  $D$  along  $K$ , i.e.  $H = R +_K D$  in (2) of Figure 7.

### 3.4 Gluing Construction and Pushout

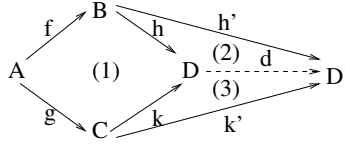
The idea of the gluing construction of graphs makes sense also for other kinds of structures, where the idea is to construct the union of structures along a common substructure. For structures given by geometrical figures this kind of union or gluing is shown in Figure 9.

In the framework of category theory the idea of the gluing construction can be formalized by the notion of a *pushout*: Given objects (e.g. sets, graphs or structures)  $A, B$ , and  $C$  and morphisms (e.g. functors, graph or structure morphisms)  $f : A \rightarrow B$  and  $g : A \rightarrow C$  an object  $D$  together with morphisms  $h : B \rightarrow D$  and  $k : C \rightarrow D$  is called *pushout* of  $f$  and  $g$  if we have  $h \circ f = k \circ g$  (i.e. diagram (1) in Figure 10 commutes) and the following universal property is satisfied:

For all objects  $D'$  and morphisms  $h' : B \rightarrow D', k' : C \rightarrow D'$  with  $h' \circ f = k' \circ g$  (i.e. the outer diagram in Figure 10 commutes) we have a unique morphism  $d : D \rightarrow D'$  s.t.  $d \circ h = h'$  and  $d \circ k = k'$  (i.e. diagrams (2) and (3) commute).



**Fig. 9.** Gluing Construction for Geometrical Figures



**Fig. 10.** Universal Pushout Property

In the category **Sets** of sets and functions the pushout object  $D$  is given by the quotient set

$$D = B + C / \equiv, \text{ short } D = B +_A C,$$

where  $B + C$  is the disjoint union of  $B$  and  $C$  and  $\equiv$  the equivalence relation generated by  $f(a) \equiv g(a)$  for all  $a \in A$ . In fact  $D$  can be interpreted as the gluing of  $B$  and  $C$  along  $A$ : Starting with the disjoint union  $B + C$  we glue together the elements  $f(a) \in B$  and  $g(a) \in C$  for each  $a \in A$ .

In the category **Graph** the pushout graph  $D$  can be constructed component-wise for the set of edges and the set of vertices using the pushout construction in **Sets** discussed above. This shows that also the pushouts in **Graph** can be interpreted as a gluing construction (see Figure 8). In general, the pushout graph  $D = (D_E, D_V, s_D, t_D, le_D, lv_D)$  is given as follows:

$$\begin{aligned} & - D_E = B_E +_{A_E} C_E \\ & - D_V = B_V +_{A_V} C_V \\ & - s_D(e) = \begin{cases} [s_B(e')] & ; \text{ if } e = h_E(e') \\ [s_C(e'')] & ; \text{ if } e = k_E(e'') \end{cases} \\ & - t_D(e) = \begin{cases} [t_B(e')] & ; \text{ if } e = h_E(e') \\ [t_C(e'')] & ; \text{ if } e = k_E(e'') \end{cases} \end{aligned}$$

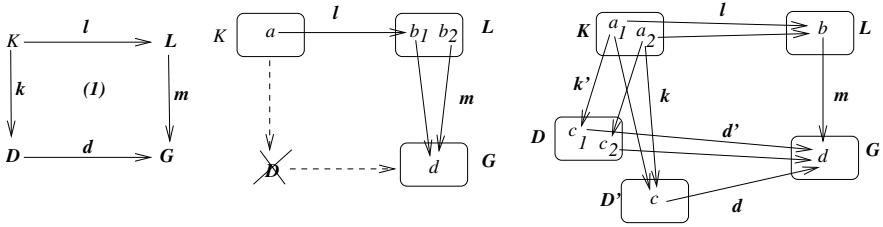
$$\begin{aligned}
 - \text{le}_D(e) &= \begin{cases} [\text{le}_B(e')] & ; \text{ if } e = h_E(e') \\ [\text{le}_C(e'')] & ; \text{ if } e = k_E(e'') \end{cases} \\
 - \text{lv}_D(v) &= \begin{cases} [\text{lv}_B(v')] & ; \text{ if } v = h_V(v') \\ [\text{lv}_C(v'')] & ; \text{ if } v = k_V(v'') \end{cases}
 \end{aligned}$$

In fact the pushout construction is well-defined and unique up to isomorphism. This means that the graph  $D$  can also be replaced by any other graph  $\overline{D}$ , which is isomorphic to  $D$ , i.e. there is a bijective graph morphism  $f : D \rightarrow \overline{D}$ .

Uniqueness of pushouts up to isomorphism is a general property of pushouts in arbitrary categories. Moreover, it is a general property that pushouts can be composed horizontally and vertically leading again to pushouts.

### 3.5 Gluing Condition and Pushout Complement

In order to construct a graph transformation from a given graph production  $p = (L \xleftarrow{l} K \xrightarrow{r} R)$  and a match  $m : L \rightarrow G$  as shown in Figure 7 we have to construct first a graph  $D$  and graph morphisms  $K \rightarrow D$  and  $D \rightarrow G$  s.t. diagram (1) in Figure 7 becomes a pushout in the category **Graph**. In this case  $D$  is called *pushout complement* of  $l : K \rightarrow L$  and  $m : L \rightarrow G$ . See also the left diagram in Figure 11. In general, however, the pushout complement may not exist, or may not be unique up to isomorphism. In Figure 11 we show two examples in the category **Sets**, where in the middle there is no pushout complement  $D$  for given functions  $l : K \rightarrow L$  and  $m : L \rightarrow G$ . On the right hand side we have two different non-isomorphic pushout complements  $D$  and  $D'$ .



**Fig. 11.** Construction, Non-Existence and Non-Uniqueness

In the category **Sets** and **Graphs** we have uniqueness of the pushout complement up to isomorphism if  $l : K \rightarrow L$  is injective. For the existence of the pushout complement we need a *Gluing Condition*. Given an injective graph morphism  $l : K \rightarrow L$  and a match  $m : L \rightarrow G$  we can construct a pushout complement  $D$  leading to the pushout (1) in Figure 11 if and only if the following *Gluing condition* is satisfied that requires that the *boundary* of the match  $m : L \rightarrow G$  is included in the gluing part  $l(K)$  of  $L$ . More formally, we have:

**Gluing Condition:**

$$\text{BOUNDARY} \subseteq \text{GLUING}$$

where *BOUNDARY* and *GLUING* are subgraphs of  $L$  defined by

- $GLUING = l(K)$
- $DANGLING = \{x \in L_V \mid \exists e \in G_E - m_E(L_E) : (m_V(x) = s_G(e) \text{ or } m_V(x) = t_G(e))\}$
- $IDENTIFICATION = \{x \in K \mid \exists y \in K : (x \neq y \text{ and } m(x) = m(y))\}$ ,  
where  $x \in K$  means  $x \in K_V$  with  $m = m_V$  or  $x \in K_E$  with  $m = m_E$ , and
- $BOUNDARY = DANGLING \cup IDENTIFICATION$

This means that the boundary of the match  $m$  given by the graph *BOUNDARY* consists of a dangling and an identification part. In the identification part we have all those nodes and edges which are identified by the match  $m$ . The dangling part consists of those nodes  $x \in L$  so that  $m_V(x)$  is adjacent to an edge  $e \in G_E$ , which is not part of the match  $m(L)$ . These edges are called *dangling edges* because they lack either the source or the target node in the set theoretical complement  $G - m(L) = (G_E - m_E(L_E), G_V - m_V(L_V))$ . For brevity we call the nodes in *DANGLING* dangling nodes.

Now we can construct the pushout complement graph  $D$  in Figure 11 in the diagram to the left by  $D = (D_E, D_V, s_D, t_D, l_{eD}, l_{vD})$  with

- $D_E = (G_E \setminus m_E(L_E)) \cup m_E(l_E(K_E))$
- $D_V = (G_V \setminus m_V(L_V)) \cup m_V(l_V(K_V))$
- $T_G = (T_N \setminus m_T(T_L)) \cup m_T(l_T(T_K))$
- $s_D, t_D, l_{eD}$ , and  $l_{vD}$  are defined by the restriction of  $s_G, t_G, l_{eG}$ , and  $l_{vG}$  respectively.

Finally the graph morphisms in the diagram to the left in Figure 11  $d : D \rightarrow G$  and  $h : K \rightarrow D$  are given by the inclusion  $D \subseteq G$  and by  $k(x) = m \circ l(x)$  for nodes and edges  $x \in K$ .

In our pacman graph grammar *PGG* considered above we can have only injective matches  $m : L \rightarrow G$ , because the pacman graph *PG* in Figure 4 and Figure 8 has no loops. This implies that the identification part of the gluing condition is always satisfied. But also the dangling part is satisfied for all productions and all matches, because all nodes of the left-hand side  $L$  of each production are gluing nodes. Hence especially all dangling nodes of  $L$  are gluing nodes. In the graph transformation shown in Figure 8 both nodes 1 and 2 are dangling and also gluing nodes.

## 4 Concepts of Parallelism

In this section we present main concepts and results for parallelism of graph transformations. We start with the concepts of parallel and sequential independence leading to a local Church-Rosser Theorem which corresponds to the concept of concurrency by interleaving. However, using the concept of parallel productions and derivations, the DPO-approach also allows to model true concurrency. The Parallelism Theorem shows equivalence of true concurrency and interleaving in our framework. Finally the Parallelism Theorem allows to formulate shift equivalence leading to canonical parallel derivations.

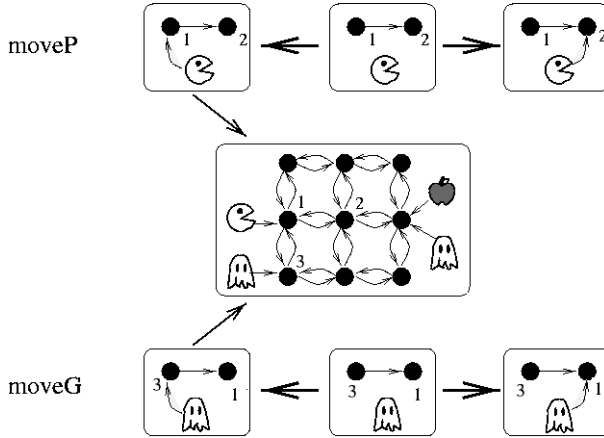
#### 4.1 Parallel and Sequential Independence

Two graph transformations  $G \Rightarrow H_1$  via  $(p_1, m_1)$ ,  $G \Rightarrow H_2$  via  $(p_2, m_2)$  are called *parallel independent* if the matches  $m_1 : L_1 \rightarrow G$  and  $m_2 : L_2 \rightarrow G$  only overlap in gluing items which are preserved by both graph transformations, i.e.

$$m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$$

for  $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$  and  $i = 1, 2$ .

In Figure 12 we show two productions  $p_1 = \text{move } P$  and  $p_2 = \text{move } G$  with matches  $m_1 : L_1 \rightarrow PG$  and  $m_2 : L_2 \rightarrow PG$  which satisfy the conditions for parallel independence. In fact, the matches overlap exactly in node 1 which is gluing node for both productions and hence preserved by the corresponding derivations. The first derivation  $PG \Rightarrow H_1$  via  $(\text{move } P, m_1)$  is explicitly shown in fig 8 with  $PG = G$  and  $H_1 = H$ . Moreover, the match  $m_2 : L_2 \rightarrow PG$  can be extended to a match  $m_2 : L_2 \rightarrow H_1$  leading to a derivation  $H_1 \Rightarrow X$  via  $(\text{move } G, m'_2)$ . In fact, the two derivations  $PG \Rightarrow H_1$  via  $(\text{move } P, m_1)$  and  $H_1 \Rightarrow X$  via  $(\text{move } G, m'_2)$  are sequential independent in the sense defined below.



**Fig. 12.** Parallel Independence

Two graph transformations  $G \Rightarrow H_1$  via  $(p_1, m_1)$  (with comatch  $m'_1 : R_1 \rightarrow H_1$ ) and  $H_1 \Rightarrow X$  via  $(p_2, m_2)$  are called *sequential independent* if the comatch  $m'_1 : R_1 \rightarrow H_1$  and the match  $m'_2 : L_2 \rightarrow H_2$  only overlap in gluing items, i.e.

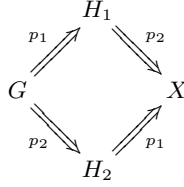
$$m'_1(R_1) \cap m_2(L_2) \subseteq m'_1(r_1(K_1)) \cap m_2(l_2(K_2))$$

Parallel and sequential independence of graph transformations are suitable conditions to allow interleaving of graph transformations as shown in the following theorem:

## 4.2 Local Church-Rosser Theorem

The following conditions for graph transformations are equivalent and each of them is leading to the diamond of parallel and sequential graph transformations in Figure 13, called *local Church-Rosser property*.

1.  $G \Rightarrow H_1$  via  $(p_1, m_1)$  and  $G \Rightarrow H_2$  via  $(p_2, m_2)$  are parallel independent
2.  $G \Rightarrow H_1$  via  $(p_1, m_1)$  and  $H_1 \Rightarrow X$  via  $(p_2, m'_2)$  are sequential independent
3.  $G \Rightarrow H_2$  via  $(p_2, m''_2)$  and  $H_2 \Rightarrow X$  via  $(p_1, m'_1)$  are sequential independent.



**Fig. 13.** Local Church-Rosser Property

An explicit proof of the local Church-Rosser Theorem is given in [14]. It is based on suitable composition and decomposition properties of pushouts.

In the following we will see that parallel independence of graph transformations also allows to construct a parallel derivation  $G \Rightarrow X$  via a parallel production  $p_1 + p_2$ .

## 4.3 Parallel Productions and Parallel Derivations

Given productions  $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$  for  $i = 1, 2$  the *parallel production*  $p_1 + p_2$  is given by

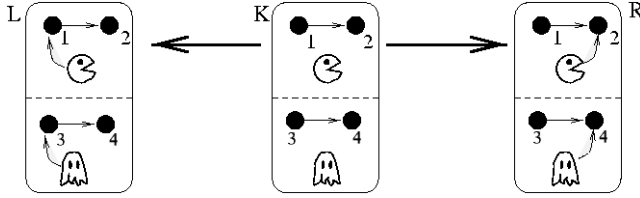
$$p_1 + p_2 = (L_1 + L_2 \xleftarrow{l_1+l_2} K_1 + K_2 \xrightarrow{r_1+r_2} R_1 + R_2)$$

where  $L_1 + L_2, l_1 + l_2$  etc. is the disjoint union of graphs and graph morphisms respectively. This corresponds to the coproduct of objects and morphisms in the category **Graphs**.

An example for the parallel production *move*  $P + \text{move } G$  is shown in Figure 14.

Parallel independence of *move*  $P$  and *move*  $G$  in Figure 12 implies according to the following Parallelism Theorem a parallel derivation  $G \Rightarrow X$  via  $(p_1 + p_2, m)$ , where the match  $m : L_1 + L_2 \rightarrow G$  is a non-injective graph morphism induced by  $m_1 : L_1 \rightarrow G$  and  $m_2 : L_2 \rightarrow G$ . The nodes 4 and 1 in Figure 14 are identified with node 1 in Figure 12. In the derived graph  $X$  pacman is on node 2 and the ghost on node 1.

In general, a derivation with a parallel production is called *parallel derivation*.



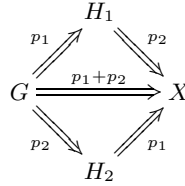
**Fig. 14.** Parallel Production *move P + move G*

#### 4.4 Parallelism Theorem

The following conditions for graph transformations are equivalent;

1.  $G \Rightarrow H_1$  via  $(p_1, m_1)$  and  $G \Rightarrow H_2$  via  $(p_2, m_2)$  are parallel independent
2.  $G \Rightarrow X$  via  $(p_1 + p_2, m)$  is a parallel derivation, where  $(p_1 + p_2)$  is the parallel production of  $p_1$  and  $p_2$  and  $m_1 : L_1 + L_2 \rightarrow G$  is the match induced by  $m : L_1 \rightarrow G$  and  $m_2 : L_2 \rightarrow G$ .

Together with the Local Church-Rosser Theorem we obtain the parallelism diamond shown in Figure 15.



**Fig. 15.** Parallelism Diamond

If  $p_1$  and  $p_2$  are sequentially independent in a derivation sequence  $G_1 \xRightarrow{p_1} G_2 \xRightarrow{p_2+p_3} G_3$  then this sequence is *shift equivalent* to a derivation sequence  $G_1 \xRightarrow{p_1+p_2} G'_2 \xRightarrow{p_3} G_3$  and we obtain the *shift relation* shown in Figure 16.

$$G_1 \xRightarrow{p_1} G_2 \xRightarrow{p_2+p_3} G_3 \quad \sqsubseteq_{\text{shift}} \quad G_1 \xRightarrow{p_1+p_2} G'_2 \xRightarrow{p_3} G_3$$

**Fig. 16.** Shift Relation

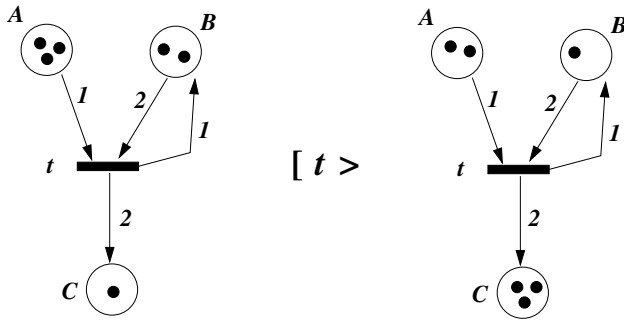
*Shift equivalence* on parallel derivations is the closure of the shift relation under parallel and sequential composition. The shift relation is well-founded. The minimal derivations with respect to shift relation are called *canonical derivations*. Canonical derivations are unique representations of shift equivalent parallel derivation classes (see [3] for more details).

## 5 Graph Grammars, Petri Nets and Concurrent Semantics

In this section we discuss the relationship between graph grammars and Petri nets. Both of them are well-known as specification formalisms for concurrent and distributed systems. First we show how the token game of place-transition nets can be modeled by double pushouts of discrete labeled graphs. This allows to relate basic notions of place-transition nets like marking, enabling, firing, steps and step sequences, to corresponding notions of graph grammars and to transfer semantical concepts from Petri nets to graph grammars. Since a marking of a net on one hand and a graph of a graph grammar on the other hand correspond to the state of a system to be modeled, graph grammars can be seen to generalize place-transition nets by allowing more structured states. In the second part of this section we give a short overview of the concurrent semantics of graph transformations presented in [3] of the Handbook of Graph grammars volume 3, which is strongly influenced by corresponding semantical constructions for Petri nets in [36]. Finally let us point out that we discuss the modification of the net structure of Petri nets using graph transformations in the next chapter.

### 5.1 Correspondence of Notions between Petri Nets and Graph Grammars

The firing of a transition in a place-transition net can be modeled by a double pushout in the category of discrete graphs labeled over the places of the transitions. Let us consider the transition firing as token game in Figure 17.



**Fig. 17.** Transition Firing as Token Game

The transition  $t$  in Figure 17 requires in the pre-domain one token on place  $A$  and two tokens on place  $B$  and produces in the post-domain one token on  $B$  and two tokens on place  $C$ . This corresponds to the production in the upper row of Figure 18, where the left hand side consists of three nodes labeled  $A, B$  and  $B$  and the right hand side of three nodes labeled  $B, C$  and  $C$ . The empty interface



of the production means that no node is preserved by the production, which corresponds to the token game in place-transition nets. In fact, the transition  $t$  in Figure 17 consumes two tokens and produces one token on place  $B$ . Preservation of tokens in the framework of Petri nets can be modeled by contextual nets, and transition with context places can be modeled by productions with nonempty interface.

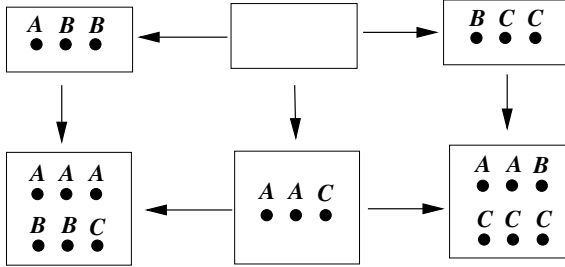


Fig. 18. Transition Firing as Double Pushout

The marking of the left-hand side net in Figure 17 corresponds to the discrete graph to the left in the lower row of Figure 18, while the marking after firing of the transition in Figure 17 corresponds to the discrete graph to the right.

The discrete graph in the middle of Figure 18 is the result of the deleting step of the double pushout and that on the right in the lower row is the result of the adding step. This shows that the firing step in Figure 17 corresponds exactly to a direct derivation in the double-pushout approach. This correspondence of notions between place/transition nets and graph grammars is shown in Table 1 in more detail. In fact, enabling of a transition at a marking corresponds to applicability of a production to a graph, concurrency of transitions corresponds to parallel independent productions applied with non-overlapping matches, conflicts correspond to parallel dependent direct derivations with overlapping matches, a parallel transition step of concurrent transitions corresponds to a parallel direct derivation, and finally a step sequence to a parallel derivation.

## 5.2 Concurrent Semantics of Graph Transformation

For sequential systems it is often sufficient to consider an input/output semantics and thus the appropriate semantic domain is usually a suitable class of functions from the input to the output domains. When concurrent or distributed features are involved, instead, typically more information about the actual computation of the system has to be recorded in the semantic domain. For instance, one may want to know which steps of computation are independent (concurrent), which are causally related and which are the (non-deterministic) choice points. This information is necessary, for example, if one wants to have a compositional semantics, allowing to reduce the complexity of the analysis of concurrent systems

**Table 1.** Correspondence of Notions

Petri Nets	Graph Grammars
tokens	nodes
places	node labels
marking	discrete, labeled graph
transition enabled at a marking	production applicable to a graph
firing	direct derivation
firing sequence	derivation
concurrent transitions	parallel independent productions
conflict	parallel dependence
step	parallel direct derivation
step sequence	parallel derivation

built from smaller parts, or if one wants to allocate a computation on a distributed architecture. Roughly speaking, *non-determinism* can be represented either by collecting all the possible different computations in a set, or by merging the different computations in a unique *branching* structure where the choice points are explicitly represented. On the other hand, *concurrent* aspects can be represented by using a *truly concurrent* approach, where the casual dependencies among events are described directly in the semantics using a partially ordered structure. Alternatively, an *interleaving* approach can be adopted, where concurrency is reduced to non-determinism, in the sense that the concurrent execution of events is represented as the non-deterministic choice among the possible interleavings of such events.

Let us first have a look to the area of Petri nets, where a well-established theory has been developed already.

Petri nets have been equipped with rich, formal computation-based semantics, including both interleaving and truly concurrent models. In many cases such semantics have been defined by using well-established categorical techniques, often involving adjunctions between suitable categories of nets and corresponding categories of models. Let us point out especially the semantics of safe place-transition nets presented as a chain of adjunctions by Winskel [36].

To propose graph transformation systems as a suitable formalism for the specification of concurrent/distributed systems that generalizes Petri nets, we are naturally led to the attempt of equipping them with a satisfactory semantic framework, where the truly concurrent behavior of grammars can be suitably described and analyzed. The basic result for interleaving and concurrent semantics of graph transformation are the local Church-Rosser Theorem and the Parallelism Theorem presented in the previous section. In the following we present the main ideas of trace, process and event structure semantics for graph transformations. For a more detailed overview we refer to the handbook article [3].

The *trace semantics* for graph transformations is based on parallel derivation sequences introduced in the previous section. Derivation traces are defined as equivalence classes of parallel derivations with respect to the shift equivalence

lence, which is the closure of the shift relation (see Figure 16) under parallel and sequential composition. Abstraction equivalence is a suitable refinement of the isomorphism relation on parallel derivations, which allows to obtain a well-defined concatenation of derivation traces. This leads to a category  $\mathbf{Tr}(\mathbf{G})$  of derivation traces of a graph grammar  $G$ , which can be considered as the trace semantics of  $G$ .

The *process semantics* for graph transformations is based on the notion of a graph process, which is a suitable generalization of a Petri net process. In fact, the idea of occurrence nets and concatenable net processes has been generalized to occurrence graph grammars and concatenable graph processes. The mapping of an occurrence graph grammar  $O$  to the original graph grammar  $G$  determines for each derivation of  $O$  a corresponding derivation of  $G$ , such that all derivations of  $O$  correspond to the full class of shift-equivalent derivations. This means that the graph process, defined by the occurrence graph grammar  $O$  together with the mapping from  $O$  to  $G$ , can be considered as an abstract representation of the shift-equivalence class. Hence the graph process plays a role similar to the canonical derivation introduced in the previous section. The process semantics for graph transformations is defined by the category  $\mathbf{CP}(\mathbf{G})$  of abstract graphs as objects and concatenable processes of  $G$  as morphisms.

The *event structure semantics* for graph transformations allows to construct an event structure for a graph grammar  $G$  which - in contrast to trace and process semantics - allows to reflect the intrinsic non-determinism of a grammar. Event structures and domains are well-known semantical models not only for Petri nets, but also for other specification techniques for concurrent and distributed systems. The domain  $\mathbf{Dom}(\mathbf{G})$  of a graph grammar is a partially ordered set, where the elements of  $\mathbf{Dom}(\mathbf{G})$  are derivation traces starting at the start graph  $G_S$  of  $G$ , and we have  $d_1 \leq d_2$  for derivation traces  $d_1 : G_S \Rightarrow G_1$  and  $d_2 : G_S \Rightarrow G_2$ , if there is a derivation trace  $d : G_1 \Rightarrow G_2$  with  $d \circ d_1 = d_2$ . Roughly spoken an event  $e$  in the event structure  $\mathbf{ES}(\mathbf{G})$  of the graph grammar  $G$  corresponds to the application of a basic production  $p(e)$  in a derivation trace  $d(e) : G_S \Rightarrow G$ . Moreover, we have a partial order  $\leq$  and a conflict relation  $\sharp$  in  $\mathbf{ES}(\mathbf{G})$ , where roughly spoken  $e_1 \leq e_2$  means  $d(e_1) \leq d(e_2)$ , and  $e_1 \sharp e_2$  means that there is no derivation trace  $d : G_S \Rightarrow G$  including both  $p(e_1)$  and  $p(e_2)$ . In the first case  $e_1$  and  $e_2$  are casually related and in the second case they are in conflict.

In the handbook article [3], where all these semantics are presented in detail, it is also shown how these different graph transformation semantics are related with each other (see 1.-3. below).

1. The trace semantics  $\mathbf{Tr}(\mathbf{G})$  and the process semantics  $\mathbf{CP}(\mathbf{G})$  are equivalent in the sense that both categories are isomorphic.
2. For consuming graph grammars  $G$  the event structure semantics  $\mathbf{ES}(\mathbf{G})$  and the domain semantics  $\mathbf{Dom}(\mathbf{G})$  are conceptually equivalent in the sense that one can be recovered from the other. A grammar  $G$  is called *consuming* if each production of the grammar deletes at least one node or edge. This correspondence is a consequence of a well-known general result concerning the equivalence of prime event structures and domains, where the configurations

of a prime event structure are the elements of the domain. A configuration of a prime event structure is a subset of events, which is left-closed and conflict free. In our case the configurations of  $\mathbf{ES}(\mathbf{G})$  correspond to the derivation traces in  $\mathbf{Dom}(\mathbf{G})$ .

3. As a consequence of results 1 and 2 above we obtain the following intuitive characterization of events and configurations from  $\mathbf{ES}(\mathbf{G})$  in terms of processes: Configurations correspond to processes, which have as source graph the start graph of the grammar. Events are one-to-one with a subclass of such processes having a production which is the maximum w.r.t. the casual ordering.
4. In the case of Petri nets Winskel has shown in [36] that the category of safe place-transition nets is related by a chain of adjoint functors to the categories of domains and prime event structures. Motivated by this chain of adjunctions Baldan has shown in his dissertation [2] that there is a chain of functors between the category of graph grammars and prime event structures, which is based on the trace and event structure semantics  $\mathbf{Tr}(\mathbf{G})$  and  $\mathbf{ES}(\mathbf{G})$  discussed above. In fact, all but one steps in this chain of functors have been shown already to be adjunctions as in the case of Petri nets.

## 6 Introduction to Petri Net Transformations

In the second part of this contribution we investigate Petri net transformations.

Note that there is a shift of paradigm. In graph transformation systems graph productions are used to model the behavior. Obviously, this is not required for Petri nets as the token game already models the behavior. In the area of Petri nets the transformations are used to describe changes of the Petri net structure. So, we can describe the stepwise development of nets, and have a formal foundation for the evolution of Petri nets. The main advantages of Petri net transformations are:

- the rule-based approach
- compatibility with structuring and marking graph semantics
- extension to refinement

There already have been a few approaches to describe transformations of Petri nets formally (e.g. in [5, 6, 33, 7, 35]). The intention has been mainly on reduction of nets to support verification, and not on the development process itself.

First we discuss briefly the formal foundation of Petri net transformations as an instantiation of so called high-level replacement systems. This is a generalization of the DPO-approach from graphs to arbitrary specification techniques, that can be instantiated especially to graphs and different classes of Petri nets (see Figure 19). Subsequently we give an extensive example, stepwise developing the baggage handling system of an airport. Finally we discuss the relevance of net transformations as means for the rule-based modification and refinement of nets.

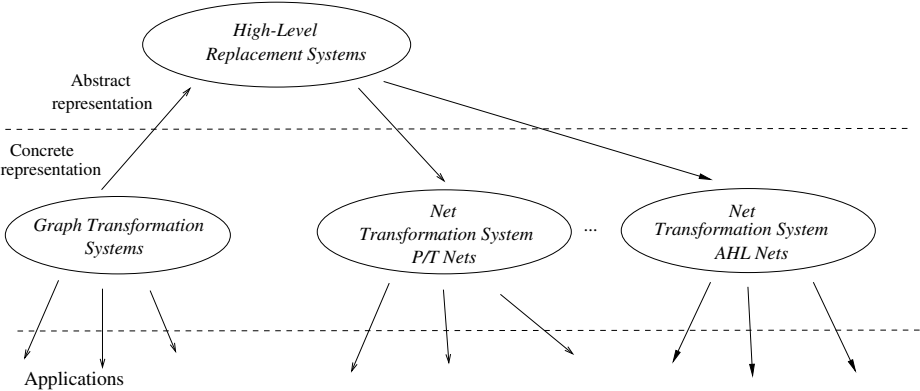


Fig. 19. Generalization and Instantiation

6.1 Formal Foundation Based on High-Level Replacement Systems

In this section we sketch the abstract frame work, that comprises the transformations of graphs in the previous and of Petri nets in the next sections. High-level replacement systems can be considered as a general description of replacement systems, where a left-hand side of the rule is replaced by a right-hand side in the presence of an interface. Historically, rules and transformations of Petri nets have been introduced as an instantiation of high-level replacement systems [12, 13, 28].

These kinds of replacement systems have been introduced in [13] as a categorical generalization of graph transformations in the DPO-approach. High-level replacement systems are formulated for an arbitrary category  $\mathbf{Cat}$  with a distinguished class  $\mathcal{M}$  of morphisms, called  $\mathcal{M}$ -morphisms. Figure 20 illustrates the main idea for some arbitrary specification or structure. The rule given in the upper line describes that a black triangle is replaced by a long dotted rectangular, if there is a light grey square below the triangle. The transformation is given by the bottom line, where the replacement specified by the rule is carried out.

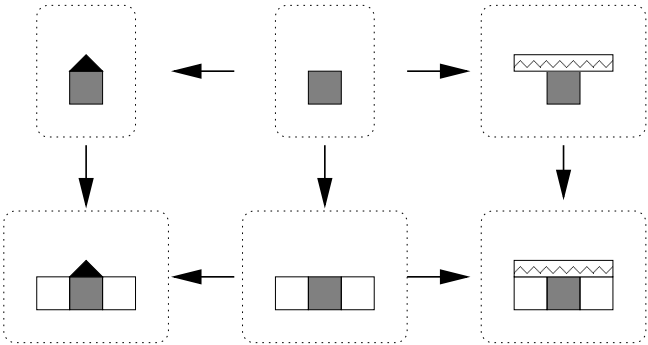


Fig. 20. Abstract Example

High-level replacement systems are a categorical generalization of the algebraic approach to graph transformation systems with double pushouts. They allow formulating the same notions as for graph transformation systems, but not only for graphs but for objects of arbitrary categories. That means, instead of replacing one graph by another one, now one object is replaced by another one. Due to the categorical formulation of high-level replacement systems the focus is not on the structure of the objects but on the properties of the category.

To achieve the results known in graph transformation systems, the instantiated category of a high-level replacement system has to satisfy certain HLR-conditions. In [24] an elegant reformulation of some HLR-conditions [26] is given in terms of adhesive categories.

In [27] we have extended the theory of high-level replacement systems where rules and transformations are required to preserve some desired properties of the specification. To do so, rules and transformations are equipped with an additional morphism that has to preserve or reflect specific properties. At this abstract level we merely can assume suitable classes of morphisms and then guarantee that these morphisms lead to property preserving rules and transformations. In Section 7.6 (and much more detailed in [29]) we give a glance how this approach works for Petri nets.

## 6.2 Example: Baggage Handling System

In this example we illustrate the rule-based modification of place/transition nets. Rules describe the replacement of a left-hand side net by a right-hand side net. The application of the rule yields a transformation where in the source net the subnet corresponding to the left-hand side is replaced by the subnet corresponding to the right-hand side. At this level as well as in this example there are no statements about the properties of the modified nets. Nevertheless based on the transformation we illustrate here, we already have developed a theory, called rule-based refinement, where the transformations are extended to introduce, preserve, or reflect net specific properties. In [29] a comprehensive survey can be found, in Section 7.6 we discuss this theory briefly.

This example concerns the sorting, screening and moving of baggage at an airport. The physical basis of the baggage handling system consists of check-in counter, conveyor belts, sorter, screening devices, a baggage claim carousel, storages, and loading stations. The conveyor belts are transportation belts, that are starting and ending at some fixed point (as check-in, sorter, loading station, baggage claim carousel, etc). The baggage handling system comprises three check-in counters, the primary sorter, the early baggage, the lost baggage as well as the unclaimed baggage storage, the secondary sorter, two loading stations, the baggage claim sorter with two baggage claim carousels and all the conveyor belts in between. Mainly, there are the following cases to handle:

1. *Check-in*: The baggage has to be moved from the check-in counter to the right loading station of a departing carrier. It has to pass a security check (screening the baggage). At the check-in the baggage items are placed manually into the transport system.

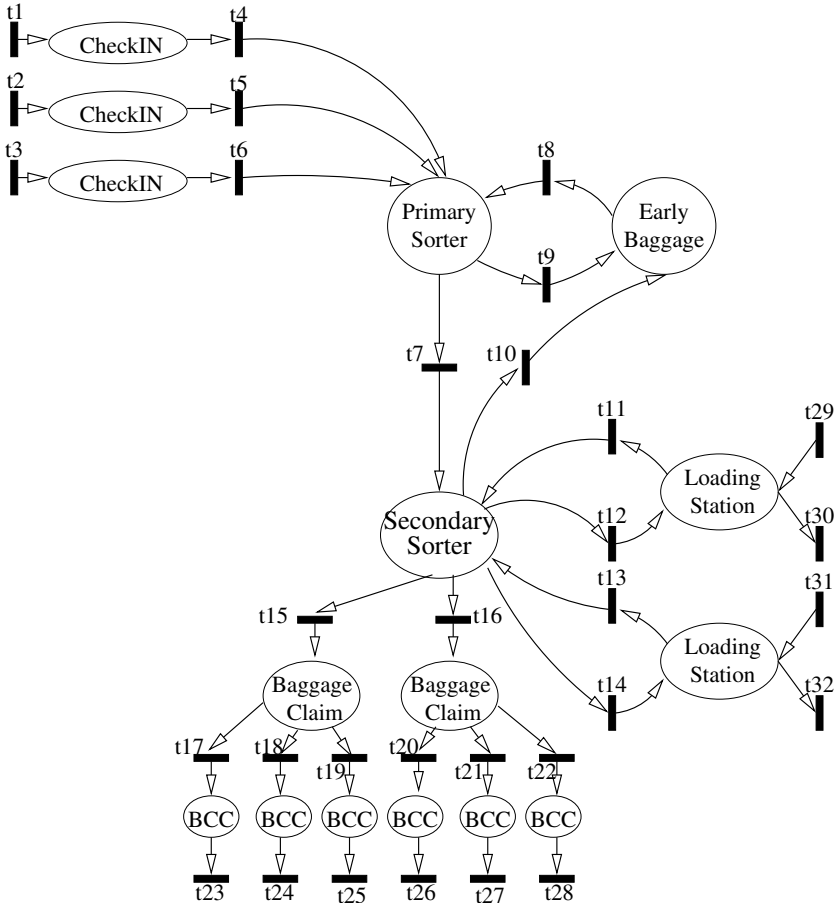
2. *Baggage Claim*: At the loading station a carrier is unloaded and the baggage items are placed manually into the transport system. The baggage has to be moved from the loading station to the right baggage claim carousel.
3. *Transfer*: The baggage has to be moved from the loading station of the arriving carrier to the loading station of the connecting flight carrier. The baggage is moved to the secondary sorter and subsequently either to the right loading station for the connecting flight or to the early baggage storage.
4. *Storing Baggage*: For baggage checked in early and for long waits between connecting flights there must be a storage, called early baggage storage. Moreover, misled or lost baggage has to be identified and is then handled manually. Baggage that is not claimed at the baggage claim carousel has to be stored as well.

To model the above given baggage handling system we can use low-level or high-level Petri nets. High-level net allow modeling the data explicitly, but for the purpose of this paper it is sufficient to use low-level nets. In fact, the basic principles for net transformations are the same for low-level and high-level nets. Subsequently we model the baggage handling system with place/transition nets, which requires some abstraction of the data – for example the baggage tags or the flight numbers are not modeled. Especially, we have modeled baggage as tokens, hence it cannot be distinguished. The choice what happens to baggage is accordingly no longer depended from the data, i.e. the baggage tag, but is done at random.

The place/transition net in Figure 21 is an abstraction of the above specified baggage handling system. The baggage handling system is an open system; baggage enters and leaves the system. We have modeled this using transitions without pre-domain for entering baggage and using transitions without post-domain for leaving baggage. Therefore we have the empty initial marking.

In the net in Figure 21 neither the conveyor belts nor the screening nor the lost or unclaimed baggage storage are modeled explicitly. Subsequently we present a step-by-step development of our first abstraction in Figure 21 that adds the lacking features. We want to add the representation of the conveyor belts by places, as well as the explicit modeling of the screening. Extending the net in this way yields a larger net. So we decompose the net into subnets in order to continue using the smaller subnets. Subsequently we introduce the subnets for the lost baggage storage and for the unclaimed baggage storage.

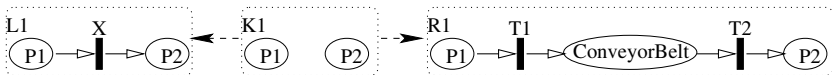
*Introducing Conveyor Belts and Screening*: The conveyor belt is not yet explicitly modeled. The transitions  $t_4$  to  $t_{14}$  represent conveyor belts. There are three different possibilities: A simple conveyor belt connecting two devices of the baggage handling system (e.g. between the sorters and the early baggage storage or the loading station) a conveyor belt connecting several devices (between check-in and the primary sorter), and a complex conveyor belt including a screening device with an optional manual check of unsafe baggage (between primary and secondary sorter). The baggage is considered by the screening device either as safe or as unsafe. If it is safe, then it is left on the conveyor belt. If it is considered



**Fig. 21.** Baggage Handling System: Net **B0**

to be unsafe, it is taken off the conveyor belt, is checked manually, and either it is taken out of the baggage handling system or it is put back into the subsequent device (sorter, storing, or loading).

For these three cases there are three rules available for the replacement of the corresponding transitions by subnets containing an explicit place **ConveyorBelt**. In the first case it is modeled by the rule  $r1 = (L1 \leftarrow K1 \rightarrow R1)$  in Figure 22. This rule states that a transition **X** is deleted (including the adjacent arcs) and is replaced by transitions **T1** and **T2** and the place **ConveyorBelt**.



**Fig. 22.** Introducing Conveyor Belts (Rule  $r1$ )

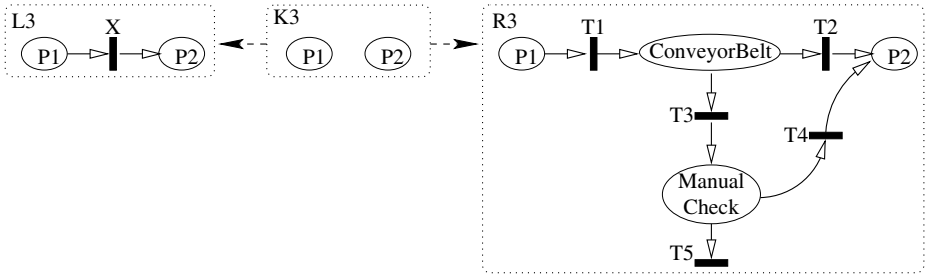


In the second case we recursively replace transitions  $\mathbf{X}$  by the already existing **ConveyorBelt** in Figure 23.



**Fig. 23.** Recursive Introduction of Conveyor Belts (Rule  $r2$ )

Introducing the screening is modeled in Figure 24 adding the new places **ConveyorBelt** and **ManualCheck** for the handling of unsafe baggage. The original transition  $\mathbf{X}$  is deleted, the two new places and the transitions in between are added. The transition  $\mathbf{T5}$  denotes the removal of the unsafe baggage.



**Fig. 24.** Introducing Screening Devices (Rule  $r3$ )

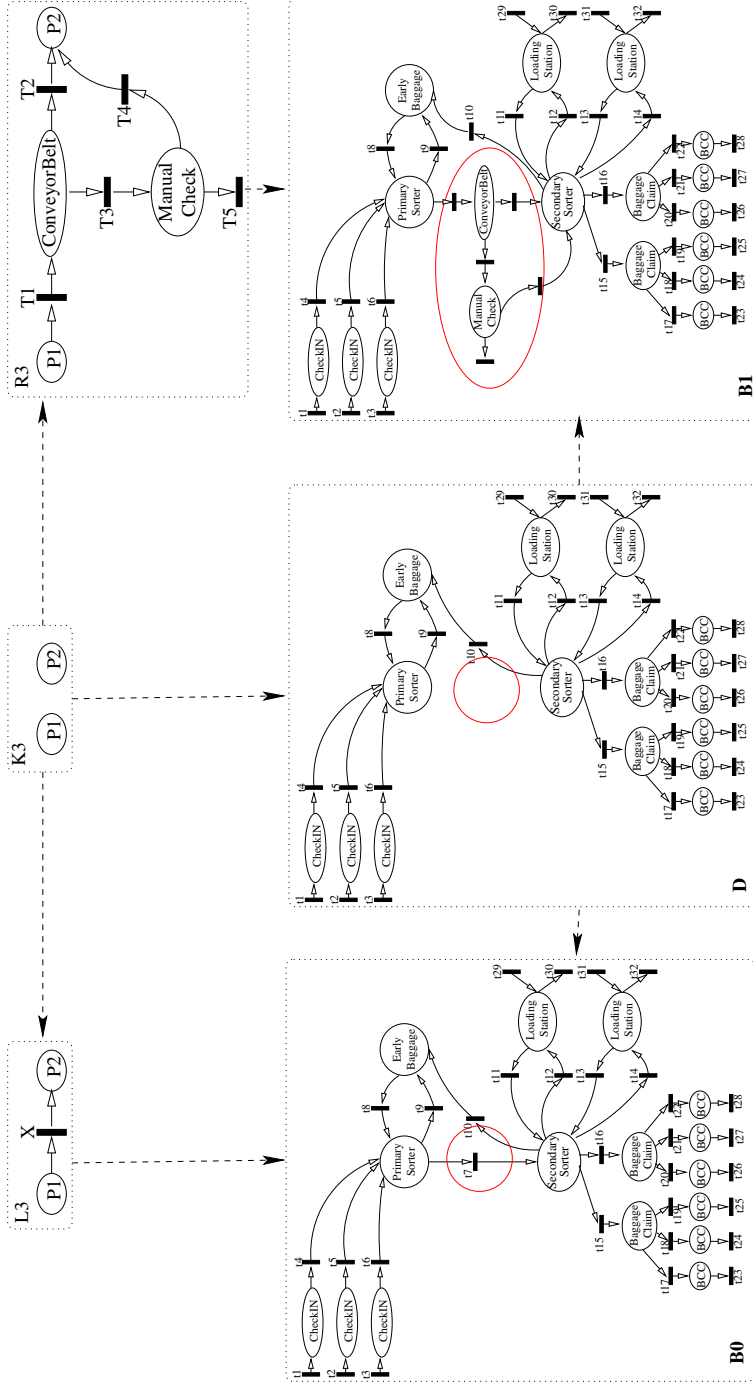
These rules can be applied several times with different matches. First we investigate the application of rule  $r3$  with match  $m$  in Figure 25 with  $m(\mathbf{X}) = \mathbf{t7}$ . Applying rule  $r3$  with match  $m$  we have again the two steps as for the application of a graph production (see 3.3):

**STEP 1 (DELETE):** Delete  $m(L3 - l(K3))$  from  $\mathbf{B0}$  leading to a context net  $\mathbf{D}$  (if the gluing condition is satisfied), s.t.  $B$  is the gluing of  $L3$  and  $D$  along  $K3$ , i.e.  $B0 = L3 +_{K3} D$  in (1) of Figure 25.

**STEP 2 (ADD):** Add  $R3 - r(K3)$  to  $D$  leading to the net  $\mathbf{B1}$ , s.t.  $\mathbf{B1}$  is the gluing of  $R3$  and  $D$  along  $K3$ , i.e.  $B1 = R3 +_{K3} D$  in (2) of Figure 25.

Then we obtain the transformation in Figure 25 consisting of two pushouts, where the context net  $\mathbf{D}$  is the net  $\mathbf{B0}$  without the transition  $\mathbf{t7}$  and the resulting net  $\mathbf{B1}$  has additional places **ConveyorBelt** and **ManualCheck** with the corresponding transitions and arcs. In Figure 25 we have indicated the changes by a light grey ellipse.

Next we apply rule  $r1$  using the matches  $m1_i : L1 \rightarrow B1$  mapping the transition  $\mathbf{X}$  to one of those transitions in  $\mathbf{B1}$  that represent a conveyor belt and mapping the places  $\mathbf{P1}$  and  $\mathbf{P2}$  the adjacent places. So, we have  $m1_i(\mathbf{X}) = \mathbf{t}_i$  for  $i \in \{8, \dots, 14\}$  leading to the nets  $B2, \dots, B8$  that are not given explicitly. At last we replace the transitions  $\mathbf{t4}$ ,  $\mathbf{t5}$ , and  $\mathbf{t6}$  by conveyor belts. We use match  $m1_4 : L1 \rightarrow B8$  with  $m1_4(\mathbf{T}) = \mathbf{t}_4$  and transform  $B8 \xrightarrow{(r1, m1_4)} B9$ . Subsequently we can apply rule  $r2$  using the matches  $m2_5(\mathbf{T}) = \mathbf{t}_5$  and  $m2_6(\mathbf{T}) = \mathbf{t}_6$ .


 Fig. 25. Transformation  $B0 \xrightarrow{(r1,m)} B1$

This results in the following transformation sequence:

$$\begin{array}{ccccccccccc} \mathbf{B0} & \xRightarrow{(r3,m)} & \mathbf{B1} & \xRightarrow{(r1,m18)} & \mathbf{B2} & \xRightarrow{(r1,m19)} & \mathbf{B3} & \xRightarrow{(r1,m110)} & \mathbf{B4} & \xRightarrow{(r1,m111)} & \mathbf{B5} & \xRightarrow{(r1,m112)} & \mathbf{B6} \\ & & & & \xRightarrow{(r1,m113)} & \mathbf{B7} & \xRightarrow{(r1,m114)} & \mathbf{B8} & \xRightarrow{(r1,m14)} & \mathbf{B9} & \xRightarrow{(r2,m25)} & \mathbf{B10} & \xRightarrow{(r2,m26)} & \mathbf{B11} \end{array}$$

Note that the nets typed bold face are illustrated in some Figure, e.g. the net **B11** is depicted in Figure 26.

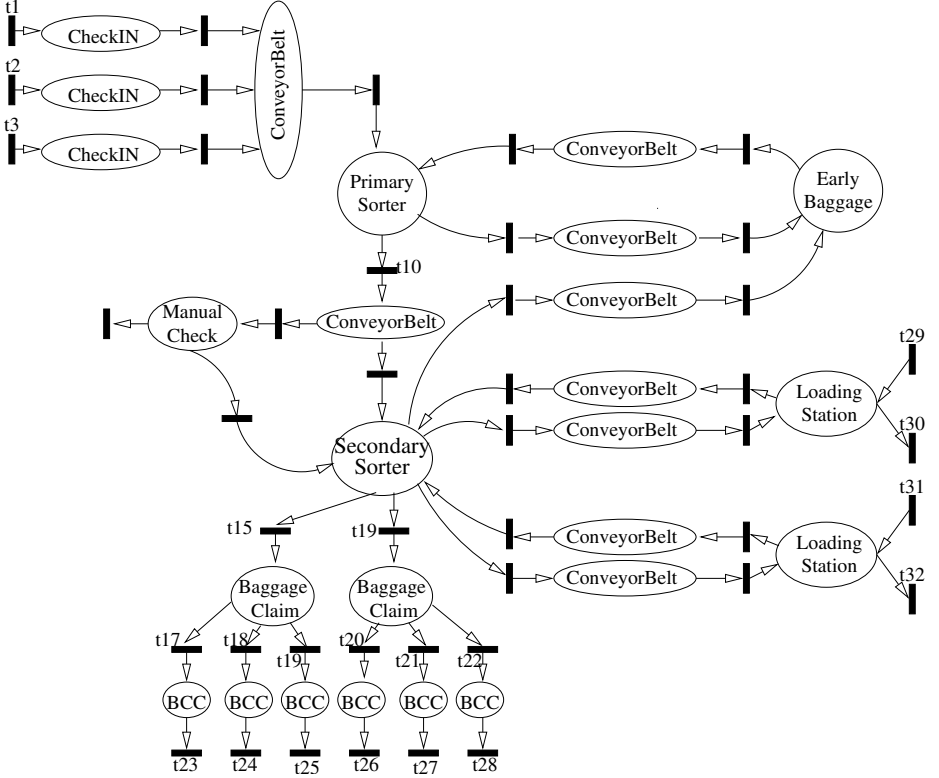
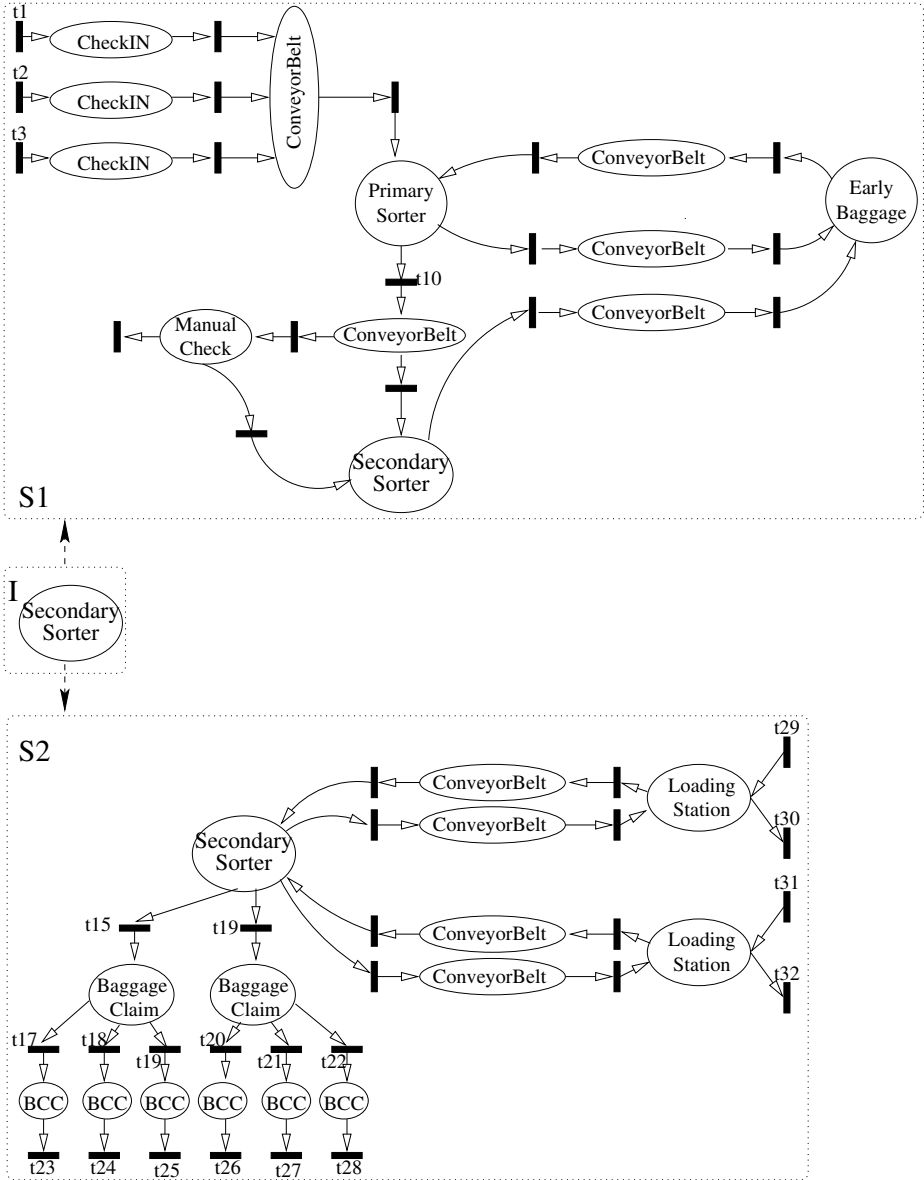


Fig. 26. Net **B11**: After Introducing all Conveyor Belts

*Decomposition of the Net:* During stepwise development a net usually reaches at some point a size, where it becomes too large and has to be decomposed.

We assume the net **B11** has become too large, so that some structuring is required. In Figure 27 the place/transition net **B11** is decomposed into two subnets **S1** and **S2** and one interface net **I**, consisting of place **SecondarySorter**. The subnets can be glued together using the union construction (see 7.4) and then yield the original net **B11**: We have the embedding of **I** into **S1** and **S2**. The union describes the gluing of the subnets along the interface, hence we have

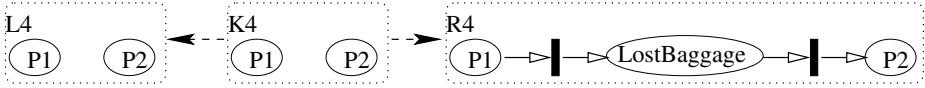


**Fig. 27.** Decomposition Using Union

the the union  $S1 +_I S2 = B11$ <sup>1</sup>. Now we can modify the subnets independently of each other provided that specific independence conditions are satisfied.

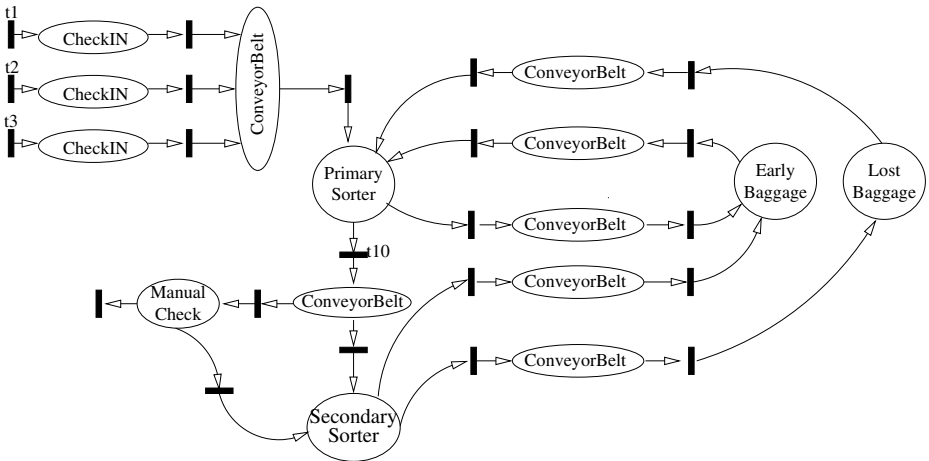
<sup>1</sup> In this case the interface net consists of one place only, so that the union corresponds to the usual place fusion of nets. But the general union construction allows having arbitrary subnets as interfaces.

*Introducing Lost Baggage Storage:* If the baggage is misled or the connecting or the departing carrier are missed, then the baggage is stored in the lost baggage storage. There it is handled manually, that is it is re-tagged and put back into the primary sorter. This is expressed at an abstract level in rule  $r4$  in Figure 28.



**Fig. 28.** Introducing Lost Baggage Storage (Rule  $r4$ )

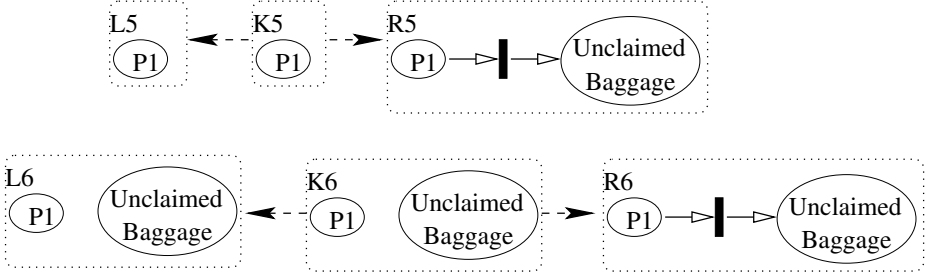
The application of rule  $r4$  to subnet **S1** using the match  $m4 : L4 \rightarrow S1$  with  $m4(\mathbf{P1}) = \mathbf{SecondarySorter}$  and  $m4(\mathbf{P2}) = \mathbf{PrimarySorter}$  yields the net  $S3$ . Applying rule  $r1$  twice, subsequently adds the corresponding conveyor belts and we have the transformation sequence  $S1 \xrightarrow{(r4, m4)} S3 \xrightarrow{r1} S4 \xrightarrow{r1} S5$ . **S5** is depicted in Figure 29.



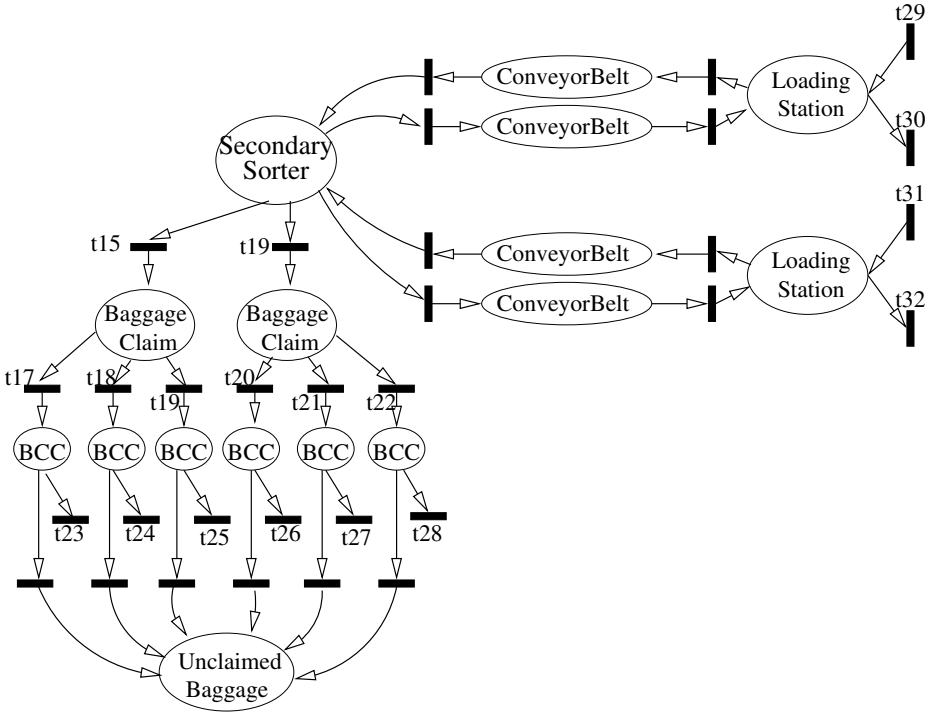
**Fig. 29.** The Resulting Subnet **S5**

*Introducing Unclaimed Baggage Storage.* If the baggage is not claimed at the baggage claim it is collected and stored in the unclaimed baggage storage. We use two rules In Figure 30 to introduce the place **UnclaimedBaggage** and the adjacent transitions recursively.

Applying first rule  $r5$  and then five times rule  $r6$  we obtain the following transformation sequence  $S2 \xrightarrow{r5} S6 \xrightarrow{5 \times (r6)} S7$ , where the resulting subnet **S7** is given in Figure 31.



**Fig. 30.** Introducing Unclaimed Baggage Storage (Rules  $r_5$  and  $r_6$ )

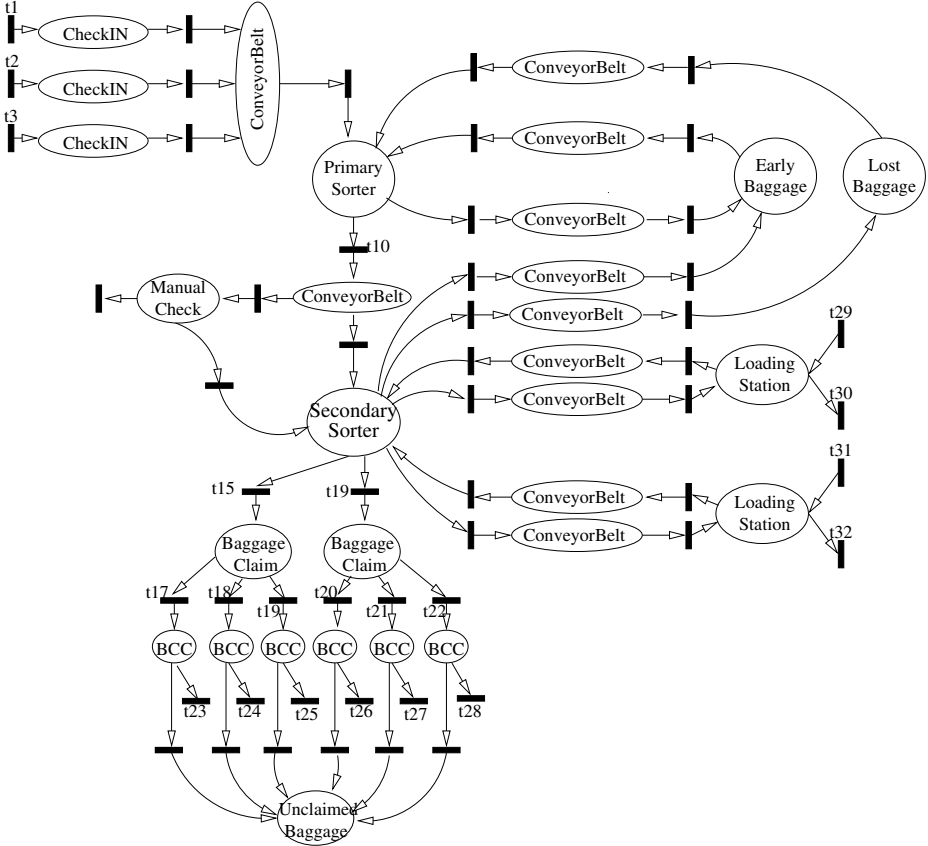


**Fig. 31.** The Resulting Subnet  $S_7$

The Union Theorem and the Parallelism Theorem together now guarantee that the resulting net **B14** in Figure 32 of the union  $S_5 +_I S_7 = B14$  is the same as the result of the following transformation sequence  $B11 \xrightarrow{r_4} B12 \xrightarrow{r_5} B13 \xrightarrow{5 \times (r_6)} B14$  according to the case without the decomposition. This is quite obvious if the interface net consists of one place only. In case of more complex interface nets this result can be only achieved if some independence conditions are satisfied. This condition states in principle that nothing from the interface net may be deleted.

### 6.3 Relevance of Petri Net Transformations

The above example illustrates only some of the possibilities and advantages of net transformations. The usual argument in favor of formal techniques, to have precise notions and valid results clearly holds for this approach as well.



**Fig. 32.** The Resulting Net **B14**

Moreover, we have already investigated net transformations in high-level Petri net classes (see Section 7.6) that are even more suitable for system modeling than the place/transition nets in our example. The impact for system development is founded in what results from net transformations:

– *Stepwise Development of Models*

The model of a complex software system may reach a size that is difficult to handle and may compromise the advantages of the (formal) model severely. The one main counter measure is breaking down the model into sub-models, the other is to develop the model top-down. In top-down development the first model is a very abstract view of the system and step by step more modeling details and functionality are added. In general however, this results

in a chain of models, that are strongly related by their intuitive meaning, but not on a formal basis.

Petri net transformations fill this gap by supporting the step-by-step development of a model formally. Rules describe the required changes of a model and their application yields the transformations of the model. Especially the repeated use of a rule ensures a uniform change of a subnet that appears as multiple copies in the model (e.g. replacing one transition by the explicit place **ConveyorBelt** and its adjacent transitions).

Moreover, the representation of change in a visual way using rules and transformations is very intuitive and does not require a deeper knowledge of the theory.

- *Distributed Development of Models*

Decomposing a model, that is too large, is an important technique for the development of complex models. To combine the advantages of a horizontal structuring with the advantages of step-by-step development techniques for ensuring the consistency of the composed model are required. Then a distributed step-by-step development is available, that allows the independent development of sub-models.

The theory of net transformations comprises horizontal structuring techniques and ensures compatibility between these and the transformations. In our example we have employed the union construction for the decomposition, and have subsequently developed the subnets independently of each other. The theory allows much more complex decompositions, where the independence of the sub-models is not as obvious as in the given example. So, the formal foundation for the distributed development of complex models is given.

- *Incremental Verification*

Pure modification of Petri nets is often not sufficient, since the net has some desired properties that have to be ensured during further development. Verification of each intermediate model requires a lot of effort and hence is cost intensive. But refinement can be considered as the modification of nets preserving desired properties. Hence the verification of properties is only required for the net, where they can be first expressed. In this way properties are introduced into the development process and are preserved from then on. Rule-based refinement modifies Petri nets using rules and transformations so that specific system properties are preserved. For a brief discussion see Section 7.6.

- *Foundation for Tool Support*

A further advantage is the formal foundation of rule-based refinement and/or rule-based modification for the implementation of tool support. Due to the theory of Petri net transformations we have a precise description, how rules and transformation work on Petri nets. Tool support is for the practical use the main precondition. The user should get tool support for defining and applying rules. The tool should assist the choice as well as the execution of rules and transformations.



– *Variations of the Development Process*

Another area, where transformations are very useful, concerns variations in the development process. Often a development is not entirely unique, but variations of the same development process lead to variations in the desired models and resulting systems. These variations can be expressed by different rules yielding different transformations, that are used during the step-by-step development. In our example we can obtain various different baggage handling systems, depending on the rules we use. We can have a system where each conveyor belt is equipped with screening device, if we always use rule  $r3$  instead of rule  $r1$ .

## 7 Concepts of Petri Net Transformations

In this section we give the precise definitions of the notions that we have already used in our example. For notions and results beyond that we give a brief survey in Section 7.6 and refer to literature.

### 7.1 Place/Transition Nets and Net Morphisms

Let us first present a notation of place/transition net that is suitable for transformations is the algebraic approach.

These nets are given in the algebraic style as introduced in [25]. A place/transition net  $N = (P, T, pre, post)$  is given by the set of places  $P$ , the set of transitions  $T$ , and two mappings  $pre, post : T \rightarrow P^\oplus$ , the pre-domain and the post-domain.

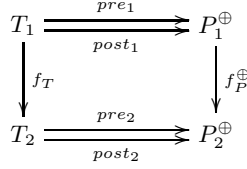
$$T \begin{array}{c} \xrightarrow{pre} \\ \xrightarrow{post} \end{array} P^\oplus .$$

$P^\oplus$  is the free commutative monoid over  $P$  that can also be considered as the set of finite multisets over  $P$ . The pre- (and post-) domain function maps each transition into the free commutative monoid over the set of places, representing the places and the arc weight of the arcs in the pre-domain (respectively in the post-domain). An element  $w \in P^\oplus$  can be presented as a linear sum  $w = \sum_{p \in P} \lambda_p \cdot p$  or as a function  $w : P \rightarrow \mathbb{N}$ . We can extend the usual operations and relations as  $\oplus$ ,  $\ominus$ ,  $\leq$ , and so on.

Based on the algebraic notion of Petri nets [25] we use simple homomorphisms that are generated over the set of places. These morphisms map places to places and transitions to transitions. The pre-domain of a transition has to be preserved, that is even if places may be identified the number of tokens that are taken, remains the same. This is expressed by the condition  $pre_2 \circ f_T = f_P^\oplus \circ pre_1$ .

A morphism  $f : N_1 \rightarrow N_2$  between two place/transition nets  $N_1 = (P_1, T_2, pre_1, post_1)$  and  $N_2 = (P_2, T_2, pre_2, post_2)$  is given by  $f = (f_P, f_T)$  with  $f_P : P_1 \rightarrow P_2$  and  $f_T : T_1 \rightarrow T_2$  so that  $pre_2 \circ f_T = f_P^\oplus \circ pre_1$  and  $post_2 \circ f_T = f_P^\oplus \circ post_1$ . The diagram schema for net morphisms is given in Figure 33.

Several examples of net morphisms can be found in Figure 25 where the dashed arrows denote injective net morphisms.

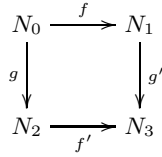
**Fig. 33.** Net Morphism

## 7.2 Rules and Transformations

The category **PT** consists of place/transition nets as objects and place/transition net morphisms as morphisms. In order formalize rules and transformations for nets in the DPO-approach we first state the construction of pushouts in the category **PT** of place/transition nets. For any span of two morphisms  $N_1 \leftarrow N_0 \rightarrow N_2$  the pushout can be constructed. The construction is based on the pushouts for the sets of transitions and sets of places in the category **Set** of sets and is similar to the pushout construction for graphs (see 3.4).

Given the morphisms  $f : N_0 \rightarrow N_1$  and  $g : N_0 \rightarrow N_2$  then the pushout  $N_3$  with the morphisms  $f' : N_2 \rightarrow N_3$  and  $g' : N_1 \rightarrow N_3$  is constructed (see Figure 34) as follows:

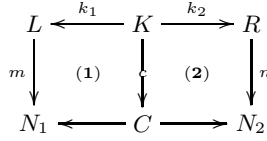
- $T_3 = T_1 +_{T_0} T_2$  with  $f'_T$  and  $g'_T$  as pushout of  $f_T$  and  $g_T$  in **Set**.
- $P_3 = P_1 +_{P_0} P_2$  with  $f'_P$  and  $g'_P$  as pushout of  $f_P$  and  $g_P$  in **Set** as well.
- $pre_3(t) = \begin{cases} [pre_1(t_1)] & \text{; if } g'_T(t_1) = t \\ [pre_2(t_2)] & \text{; if } f'_T(t_2) = t \end{cases}$
- $post_3(t) = \begin{cases} [post_1(t_1)] & \text{; if } g'_T(t_1) = t \\ [post_2(t_2)] & \text{; if } f'_T(t_2) = t \end{cases}$

**Fig. 34.** Pushout of Nets

We introduce rules, that correspond to graph productions in the DPO-approach. Rules describe the replacement of the left-hand side net by the right-hand side net in the presence of an interface net.

- A rule  $r = (L \xleftarrow{k_1} K \xrightarrow{k_2} R)$  consists of place/transition nets  $L$ ,  $K$  and  $R$ , called left-hand side, interface and right-hand side net respectively, and two injective net morphisms  $K \xrightarrow{k_1} L$  and  $K \xrightarrow{k_2} R$ .

- Given a rule  $r = (L \xleftarrow{k_1} K \xrightarrow{k_2} R)$  a direct transformation  $N_1 \xRightarrow{r} N_2$ , from  $N_1$  to  $N_2$  is given by two pushout diagrams (1) and (2) in Figure 35. The morphisms  $m : L \rightarrow N_1$  and  $n : R \rightarrow N_2$  are called match and comatch, respectively. The net  $C$  is called pushout complement or the context net.



**Fig. 35.** Net Transformation

The illustration of a transformation can be found for our example in Figure 25, where the rule  $r1$  is applied to the net **B0** with match  $m$ . The first pushout denotes the gluing of the nets  $L3$  and  $D$  along the net  $K3$  resulting in net **B0**. The second pushout denotes the gluing of net  $R3$  and net  $D$  along  $K3$  resulting in net  $B1$ .

### 7.3 Gluing Condition and the Construction of the Context Net

Given a rule  $r$  and a match  $m$  as depicted in Figure 35, then we construct in a first step the pushout complement provided the gluing condition holds. This leads to the pushout (1) in Figure 35. In a second step we construct the pushout of  $c$  and  $k_2$  leading to  $N_2$  and the pushout (2) in Figure 35.

The gluing condition correspond exactly to the gluing condition in the graph case (see 3.5). Using the same interpretation as in the graph case, but the notation from Figure 35 we have the following:

#### Gluing Condition for Nets:

$$BOUNDARY \subseteq GLUING$$

where  $BOUNDARY$  and  $GLUING$  are subnets of  $L$  defined by

- $GLUING = k_1(K)$
- $DANGLING = \{p \in P_L \mid \exists t \in T_1 - m_T(T_L) : (m_P(p) \in pre_1(t) \text{ or } m_P(p) \in post_1(t))\}$   
where the notation  $p \in pre_1(t)$  means  $pre_1(t) = \sum_{p \in P_1} \lambda_p \cdot p$  with  $\lambda_p > 0$ , similar for  $post_1$ ,
- $IDENTIFICATION = \{x \in K \mid \exists y \in K : (x \neq y \text{ and } m(x) = m(y))\}$ ,  
where  $x \in K$  means  $x \in P_K$  with  $m = m_P$  or  $x \in T_K$  with  $m = m_T$ , and
- $BOUNDARY = DANGLING \cup IDENTIFICATION$

Now the context net  $C$  is the pushout complement  $C$  in Figure 35 that is constructed by:

- $P_C = (P_1 \setminus m_P(P_L)) \cup m_P(k_{1P}(P_K))$
- $T_C = (T_1 \setminus m_T(T_L)) \cup m_T(k_{1T}(T_K))$
- $pre_C = pre_{1|T_C}$  and  $post_C = post_{1|T_C}$

Note that the pushout complement  $C$  leads to the pushout (1) in Figure 35 and that it is unique up to isomorphism.

In our example of the development of the baggage handling system the gluing condition is satisfied in all cases, since the matches are all injective and places are not deleted by our rules.

## 7.4 Union Construction

The union of two Petri nets sharing a common subnet, that may be empty, is defined by the pushout construction for nets.

The union of place/transition nets  $N_1, N_2$  sharing an interface net  $I$  with the net morphisms  $f : I \rightarrow N_1$  and  $g : I \rightarrow N_2$  is given by the pushout (1) in Figure 36. Subsequently we use the short notation  $N = N_1 +_I N_2$  or  $N_1, N_2 \xRightarrow{I} N$ .

$$\begin{array}{ccc} I & \xrightarrow{f} & N_1 \\ g \downarrow & (1) & \downarrow g' \\ N_2 & \xrightarrow{f'} & N \end{array}$$

**Fig. 36.** Union of Nets

In our example we use the union construction to describe the decomposition in Figure 27. The interface net  $I$  is mapped by morphisms to the subnets  $S1$  and  $S2$ .

## 7.5 Union Theorem

The Union Theorem states the compatibility of union and net transformations: Given a union  $N_1 +_I N_2 = N$  and net transformations  $N_1 \xRightarrow{r_1} M_1$  and  $N_2 \xRightarrow{r_2} M_2$  then we have a parallel rule  $r_1 + r_2$  (analogously to a parallel production, see 4.3) and a parallel net transformation  $N \xRightarrow{r_1+r_2} M$ .  $M = M_1 +_I M_2$  is then the union of  $M_1$  and  $M_2$  with the shared interface  $I$ , provided that the given net transformations preserve the interface  $I$ .

The Union Theorem is illustrated in Figure 37:

$$\begin{array}{ccc} N_1, N_2 \xRightarrow{I} N & & \\ \downarrow r_1, r_2 & (=) & \downarrow r_1 + r_2 \\ M_1, M_2 \xRightarrow{I} M & & \end{array}$$

**Fig. 37.** Diagram for the Union Theorem

Note that the compatibility requires an independence condition stating that nothing from the interface net  $I$  may be deleted by one of the transformations of the subnets. This is obviously the case in our example, since the interface consists of one place only and the rules do not delete any places.

## 7.6 Further Results

We briefly introduce the main net classes we have studied up to now, and subsequently we present some main results.

- Place/transition nets in the algebraic style have already been introduced in the previous Section.
- Coloured Petri nets [20–22] are widely known and very popular. Their practical relevance is very high, due to the very successful tool Design/CPN [19].
- Algebraic high-level nets are available in quite a few different notions e.g. [34, 30, 28]. We use a notion that reflects the paradigm of abstract data types into signature and algebra. An algebraic high-level net (as in [28]) is given by  $N = (SPEC, P, T, pre, post, cond, A)$ , where  $SPEC = (S, OP, E)$  is an algebraic specification in the sense of [16],  $P$  is the set of places,  $T$  is the set of transitions,  $pre, post : T \rightarrow (T_{OP}(X) \times P)^\oplus$  are the pre- and post-domain mappings,  $cond : T \rightarrow \mathcal{P}_{fin}(EQNS(SIG))$  are the transition guards, and  $A$  is a  $SPEC$  algebra.

*Horizontal Structuring.* Union and fusion are two categorical structuring constructions for place/transition nets, that merge two subnets or two different nets into one.

The Union is introduced in the previous section. Now let us consider fusion: Given a net  $F$  that occurs in two copies in the net  $N_1$ , represented by two

morphisms  $F \xrightarrow[f']{f} N_1$  the fusion construction leads to a net  $N_2$ , where both occurrences of  $F$  in  $N_1$  are merged. If  $F$  consists of places  $p_1, \dots, p_n$  then each of the places occurs twice in net  $N_1$ , namely as  $f(p_1), \dots, f(p_n)$  and  $f'(p_1), \dots, f'(p_n)$ .  $N_2$  is obtained from net  $N_1$  fusing both occurrences  $f(p_i)$  and  $f'(p_i)$  of each place  $p_i$  for  $1 \leq i \leq n$ .

The Union Theorem is presented in the previous section. The Fusion Theorem [27] is expressed similarly: Given a rule  $r$  and a fusion  $F \xrightarrow{\quad} N_1$  then we obtain the same result whether we derive first  $N_1 \xrightarrow{r} N'_1$  and then construct the fusion  $F \xrightarrow{\quad} N'_1$  resulting in  $N'_2$  or whether we construct the fusion  $F \xrightarrow{\quad} N_1$  first, resulting in  $N_2$  and then perform the transformation step  $N_2 \xrightarrow{r} N'_2$ . Similar to the Union Theorem a certain independence condition is required. Both theorems state that Petri nets transformations are compatible with the corresponding structuring technique under suitable independence conditions. Roughly spoken these conditions guarantee that the interface net  $I$  and respectively the fusion net  $F$  are preserved by all net transformations.

*Parallelism.* In Section 4 the concepts of parallelism have been discussed for graphs. The main theorems hold for Petri net transformations as well.

The Church-Rosser Theorem states a local confluence in the sense of formal languages. The required condition of parallel independence means that the matches of both rules overlap only in parts that are not deleted. Sequential independence means that those parts created by the first transformation step are not deleted in the second. The Parallelism Theorem states that sequential or parallel independent transformations can be carried out either in arbitrary sequential order or in parallel. In the context of step-by-step development these theorems are important as they provide conditions for the independent development of different parts or views of the system. More details for horizontal structuring or parallelism are given in see [28] or [27].

*Refinement.* The extension of High-level replacement systems to rules and transformations preserving properties has the following impact on Petri nets: Rule-based refinement comprises the transformation of Petri nets using rules while preserving certain net properties. For Petri nets the desired properties of the net model can be expressed, e.g in terms of Petri nets (as liveness, boundedness etc.), in terms of logic (e.g. temporal logic, logic of actions etc.) in terms of relation to other models (e.g. bisimulation, correctness etc.) and so on. We have investigated the possibilities to preserve liveness of Petri nets and safety properties in the sense of temporal logic.

Summarizing, we have for place/transition nets, algebraic-high level nets and Coloured Petri nets the following results for rule-based refinement presented in table 2. For more details see [29].

## 8 Conclusion

In the first part of this paper (Sections 2 - 5) we have given a tutorial introduction to the basic notions of graph grammars and graph transformations including the relationship to corresponding notions of Petri nets. In the second part (Section 6 and Section 7) we have shown how to use Petri nets transformations for the stepwise development of systems and have included a detailed example of a baggage handling system. The main idea of Petri transformations is to extend the classical theory of Petri nets by a rule-based technique that allows studying the changes of the Petri net structure.

In our general overview of graph grammars and transformations in Section 2 we have already pointed out that there is a large variety of different approaches and application areas. The practical use of graph transformations is supported by several tools. The algebraic approach to graph transformations (presented in Sections 3 - 5) is especially supported by the graph transformation environment AGG (see the homepage of [1]). AGG includes an editor for graphs and graph grammars, a graph transformation engine, and a tool for the analysis of graph transformations. the AGG system as well as some other tools are available on a CD which is part of volume 2 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [10]. This volume provides also

**Table 2.** Achieved results

Notion/Results	PT-nets	AHL-nets	CPNs
Rules, Transformations	✓	✓	✓
Safety property preserving transformations with transition-gluing morphisms	✓	✓	✓
place-preserving morphisms	✓	✓	✓
Safety property introducing transformations	✓	✓	✓
Liveness preserving transformations	✓	?	?
Liveness introducing transformations	✓	?	?
Church Rosser I + II Theorem	✓	✓	✓
Parallelism Theorem	✓	✓	✓
Union	✓	✓	✓
Fusion	✓	✓	✓
Union Theorems I+II	✓	✓	✓
Fusion Theorem	✓	✓	✓

an excellent introduction to several application areas for graph transformations. Concurrency aspects of graph grammars, which are briefly discussed in Section 5, are presented in much more detail in volume 3 of the handbook [3]. This volume includes also an introduction to high-level replacement systems with application to algebraic specification and Petri nets including the theoretical foundations of Petri net transformations [11].

On top the graph transformation system AGG there is the GENGED environment (see the homepage of [18]) that supports the generic description of visual modeling languages for the generation of graphical editors and the simulation of the behavior of visual models. Especially, Petri net transformations can be expressed using GENGED, e.g. for the animation of Petri nets [9, 4]. In this framework, the animation view of a system modeled as a Petri net consists of a domain-specific layout and an animation according to the firing behavior of the Petri net. This animation view can be coupled to other Petri net tools [8] using the Petri Net Kernel [23] a tool infrastructure for editing, simulating and analyzing Petri nets of different net classes and for integration of other Petri net tools.

## References

1. AGG Homepage. <http://tfs.cs.tu-berlin.de/agg>.
2. P. Baldan. *Modelling Concurrent Computations: From Contextual Petri Nets to Graph Grammars*. PhD thesis, University of Pisa, 2000.

3. P. Baldan, A. Corradini, U. Montanari, F. Rossi, H. Ehrig, and M. Löwe. Concurrent Semantics of Algebraic Graph Transformations. In G. Rozenberg, editor, *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.
4. R. Bardohl and C. Ermel. Scenario Animation for Visual Behavior Models: A Generic Approach Applied to Petri Nets. In G. Juhas and J. Desel, editors, *Proc. 10th Workshop on Algorithms and Tools for Petri Nets (AWPN'03)*, 2003.
5. G. Berthelot. Checking properties of nets using transformations. *Advances in Petri Nets 1985*, Lecture Notes in Computer Science 222: pages 19–40. Springer 1986.
6. G. Berthelot. Transformations and decompositions of nets. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets*, Lecture Notes in Computer Science 254, pages 359–376. Springer, 1987.
7. R. David and H. Alla, editors. *Petri Nets and Grafcet*. Prentice Hall (UK), 1992.
8. C. Ermel, R. Bardohl, and H. Ehrig. Specification and implementation of animation views for Petri nets. In DFG Research Group *Petri Net Technology, Proc. of 2nd International Colloquium on Petri Net Technology for Communication Based Systems*, 2001.
9. C. Ermel, R. Bardohl, and H. Ehrig. Generation of animation views for Petri nets in GENGED. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems*, Lecture Notes in Computer Science 2472. Springer, 2003.
10. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.
11. H. Ehrig, M. Gajewsky, and F. Parisi-Presicce. *High-level replacement systems with applications to algebraic apecifications and Petri nets*, chapter 6, pages 341–400. Number 3: Concurrency, Parallelism, and Distribution in Handbook of Graph Grammars and Computing by Graph Transformations. World Scientific, 1999.
12. H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.
13. H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.
14. H. Ehrig. Introduction to the algebraic theory of graph grammars (A survey). In *Graph Grammars and their Application to Computer Science and Biology*, pages 1–69. Lecture Notes in Computer Science 73. Springer, 1979.
15. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific, 1999.
16. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985.
17. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
18. GenGED Homepage. <http://tfs.cs.tu-berlin.de/genged>.
19. K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN. A Reference Manual*. Meta Software Cooperation, 125 Cambridge Park Drive, Cambridge Ma 02140, USA, 1991.



20. K. Jensen. *Coloured Petri nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1992.
21. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 2: Analysis Methods. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1994.
22. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 3: Practical Use. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1997.
23. E. Kindler and M. Weber. The Petri net kernel – an infrastructure for building Petri net tools. *Software Tools for Technology Transfer*, 3(4):486–497, 2001.
24. S. Lack and P. Sobociski. Adhesive categories. In *Proc. FOSSACS 04*, 2004. to appear.
25. J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, 88(2):105–155, 1990.
26. J. Padberg. Survey of high-level replacement systems. Technical Report 93-8, Technical University of Berlin, 1993.
27. J. Padberg. Categorical approach to horizontal structuring and refinement of high-level replacement systems. *Applied Categorical Structures*, 7(4):371–403, December 1999.
28. J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.
29. J. Padberg and M. Urbášek. Rule-based refinement of Petri nets: A survey. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems*, Lecture Notes in Computer Science 2472. Springer, 2003.
30. W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, 1991.
31. L. Ribeiro, H. Ehrig, and J. Padberg. Formal development of concurrent systems using algebraic high-level nets and transformations. In *Proc. VII Simpósio Brasileiro de Engenharia de Software*, pages 1–16, Tech-report no. 93-13, TU Berlin, 1993.
32. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
33. Vanio M. Savi and Xiaolan Xie. Liveness and boundedness analysis for petri nets with event graph modules. In Jensen, K., editor, *13th International Conference on Application and Theory of Petri Nets 1992, Sheffield, UK*, Lecture Notes in Computer Science 616, pages 328–347. Springer, 1992.
34. J. Vautherin. Parallel system specification with coloured Petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 87*, pages 293–308. Lecture Notes in Computer Science 266. Springer Verlag, 1987.
35. W.M.P. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets*, Lecture Notes in Computer Science 1248, pages 407–426. Springer, 1997.
36. G. Winskel. Petri nets, algebras, morphisms, and compositionality. *Information and Computation*, 72:197–238, 1987.