# Defining Operational Behavior of Object Specifications by Attributed Graph Transformations *

Annika Wagner[1] and Martin Gogolla[2]

[1] TU Berlin; Fachbereich 13; Franklinstr. 28/29; D-10587 Berlin
[2] TU Braunschweig; Informatik, Abt. Datenbanken; Postfach 3329; D-38023 Braunschweig

**Abstract.** A single pushout approach to the transformation of attributed partial graphs based on categories of partial algebras and partial morphisms is introduced. A sufficient condition for pushouts in these categories is presented. As the synchronization mechanism we use amalgamation of rules and show how synchronization can be minimized. We point out how the results obtained can be employed in order to define an operational semantics for object specification languages.

## 1 Introduction

Graphs and graph grammars usually yield intuitive descriptions of complex phenomena in computer science. Therefore, numerous approaches to graph grammars have been put forward, among them the logical approach [6], the set theoretic approach [29], and the algebraic approach [9]. Graph-based techniques have for instance been successfully applied in the realm of software engineering development environments [13, 14], for object-oriented languages based on asynchronous communication [22, 24, 20, 21] and in logic programming [5, 28]. Within the algebraic (or categorical) school the classical double pushout approach (c.f. [9], among many others) has been accompanied by the single pushout approach [31, 25, 26].

We combine the theory [2, 32] and specification [1, 3] of partial algebras with a single pushout approach to graph grammars [26] and apply the results obtained to the specification of conceptual objects as they occur in information system design traditionally employing semantic data models [33, 19, 30].

Our interest in applying the results focusses on the object description language TROLL *light* [4, 16, 18]. The very basic idea of TROLL *light* is the uniform and coherent specification of both structure and behavior of objects. The underlying semantic model of TROLL *light* is a transition system or (as we call it) an object community. In the single states, information about currently existing objects is represented, and state transitions are caused by occurrence of finite sets of events. The aim of TROLL *light* is to model information systems and not systems in generel. Therefore we do not take into account what is called in the literature true concurrency but we are satisfied with a simpler semantic domain in favour of ease of concepts. We concentrate on TROLL *light* but the results achieved are applicable to other related object description languages like OBLOG [34], TROLL [23], or CMSL [36] and even to other languages also following the object paradigm like MONDEL [35], ALBERT [7], or $\Pi$ [15].

* Work reported here has been partly supported by by the CEC under Grant No. 6112 (COMPASS) and Grant No. 7183 (COMPUGRAPH) and by BMFT under Grant No. 01 IS 203 D (KORSO).

The structure of the paper is as follows. In the next section we sketch the concepts of the language TROLL *light* by means of an example. In Sect. 3 we explain the basic ideas and results about representing object community states as partial attributed graphs. In Sect. 4 we show how the amalgamation technique for graph grammar rules can be extended to our case, and in Sect. 5 we apply this amalgamation technique to the evolution of TROLL *light* object communities. Sect. 6 presents some conclusions and future work to be done.

## 2 A Sketch of TROLL *light*

TROLL *light* is a language for describing static and dynamic properties of objects. This is achieved by offering language features to specify object structure as well as object behavior. The main advantage of following the object paradigm is the fact that all relevant information concerning one object can be found within one single unit and is not distributed over a variety of locations. Object descriptions are called templates in TROLL *light*. Because of their pure descriptive nature templates may roughly be compared with the notion of class found in object-oriented programming languages. In the context of databases however, classes are also associated with class extensions so that we used a different notion. Templates show the following structure.

```
TEMPLATE name of the template
  DATA TYPES   data types used in current template
  TEMPLATES    other templates used in current template
  SUBOBJECTS   slots for sub-objects
  ATTRIBUTES   slots for attributes
  EVENTS       event generators
  CONSTRAINTS  restricting conditions on object states
  VALUATION    effect of event occurrences on attributes
  DERIVATION   rules for derived attributes
  INTERACTION  synchronization of events in different objects
  BEHAVIOR     description of object behavior by a CSP-like process
END TEMPLATE
```

Roughly speaking, the `DATA TYPES` and `TEMPLATES` sections are the interfaces to other templates, the `SUBOBJECTS`, `ATTRIBUTES`, and `EVENTS` sections constitute the template signature, and in the remaining sections axioms concerning static (`CONSTRAINTS` and `DERIVATION`) and dynamic (`VALUATION`, `INTERACTION`, and `BEHAVIOR`) properties are specified.

We abstract from some peculiarities of TROLL *light*, and therefore concentrate on very essential features of object description languages. Let us introduce the main ideas of the language by means of an example. We assume that objects of type author are to be modeled. For every author the name, the date of birth, and the number of books sold have to be stored (`ATTRIBUTES` section). These attributes may be changed by general state modifying operations (`EVENTS` and `VALUATION` sections). An author may change her name only once in her life (`BEHAVIOR` section). An appropriate TROLL *light* specification would look as follows:

```
TEMPLATE Author
  DATA TYPES   String, Date, Nat;
```

```
  ATTRIBUTES    Name:string; DateOfBirth:date; NumOfBooks:nat;
  EVENTS        BIRTH create(Name:string, DateOfBirth:date);
                      changeName(NewName:string);
                      addBook;
                DEATH destroy;
  VALUATION     [create(N,D)] Name=N, DateOfBirth=D, NumOfBooks=0;
                [changeName(N)] Name=N;
                [addBook ] NumOfBooks=NumOfBooks+1;
  BEHAVIOR      PROCESS authorlife1 =
                  ( addBook -> authorlife1 |
                    changeName -> authorlife2 |
                    destroy -> POSTMORTEM );
                PROCESS authorlife2 =
                  ( addBook -> authorlife2 |
                    destroy -> POSTMORTEM );
                ( create -> authorlife1 );
END TEMPLATE;
```

For templates we employ the following naming convention: Template names are written capitalized, and each template (`Author`) induces a corresponding object sort written exactly as the template but with a starting lower case letter (`author`). In an analogous way to authors, we could specify books. The important new concept here is the use of an object-valued attribute which in this case "points" from the book to the author of the book.

```
TEMPLATE Book
  DATA TYPES    String;
  TEMPLATES     Author;
  ATTRIBUTES    Author:author; Title:string;
  EVENTS        BIRTH create(Author:author, Title:string);
                DEATH destroy;
  VALUATION     [create(A,T)] Author=A, Title=T;
END TEMPLATE;
```

Up to now we have only described more or less single objects. Now we put these single objects into the bigger context of a library. Therefore we introduce books and authors as sub-objects of a library (`SUBOBJECTS` section). Within a library, an author is identified by a natural number (parameter of `Authors`) and a book by its author and its title (parameters of `Books`). Library objects may communicate with its sub-objects by calling their events (`INTERACTION` section). For instance, the specification `newAuthor(N,S,D) >> Authors(N).create(S,D)` means that whenever the event `newAuthor(N,S,D)` occurs in a library, then also the event `create(S,D)` has to occur in the author object determined by `Authors(N)`. Sets of events which are complete w.r.t. this calling mechanism are called *closed event sets*. They induce the transitions between object community states. In order to avoid contradictions closed event sets are restricted to contain only one event per object.

```
TEMPLATE Library
  DATA TYPES    String, Date, Nat;
  TEMPLATES     Author, Book;
```

```
  SUBOBJECTS    Authors(No:nat):author;
                Books(Author:author, Title:string):book,
  ATTRIBUTES    NumberOfBooks, NumberOfAuthors:nat;
  EVENTS        BIRTH create;
                      newAuthor(No:nat, Name:string, DateOfBirth:date);
                      changeAuthorName(Author:author, NewName:string);
                      newBook(Author:author, Title:string);
                      removeAuthor(Author:author);
  VALUATION     [create] NumberOfBooks=0, NumberOfAuthors=0;
                [newAuthor] NumberOfAuthors=NumberOfAuthors+1;
                [newBook] NumberOfBooks=NumberOfBooks+1;
                [removeAuthor] NumberOfAuthors=NumberOfAuthors-1;
  INTERACTION   newAuthor(N,S,D) >> Authors(N).create(S,D);
                changeAuthorName(A,N) >> A.changeName(N);
                newBook(A,T) >> Books(A,T).create(A,T),
                                A.addBook;
                removeAuthor(A) >> A.destroy;
END TEMPLATE;
```

In the `Library` template we have, for instance, the event `changeAuthorName` which triggers the `changeName` event in an author object. This does not affect the identity of `Author` objects since within `Library` they are identified by a natural number. This also does not affect the object-valued attribute `Author` in template `Book`.

We have concentrated here on the essential features of TROLL *light* which can be found also in other object description languages. The TROLL *light* features we have not mentioned in detail are means to give restricting conditions on object states (`CONSTRAINTS`) and means to specify derived attributes (`DERIVATION`). Here, we can only give a glimpse of the language. More details can be found in [4, 16, 18].

## 3 Transformation of Partial Attributed Graphs by Single Pushouts

The algebraic approach to graph transformation models graphs and graph-like structures as special types of algebras. Rewrite rules and occurrences are described by morphisms [9, 26]. The transformation concept is based on a double or single pushout, resp. But algebras can also be used as semantic domain of the specification of data types [11, 8, 12, 37]. Graphs which come equipped with a data type component are called attributed graphs. The reason for introducing them is that well-known data types need not artificially be coded into graphical structures. In terms of total algebras attributed graphs have been modeled as a combination of one algebra for the graphical part and another algebra for the data type component [27]. Hence a transformation step consists of the transformation of the graphical part, the transformation of the data part and a relating step where the attributions are computed.

In our approach we switch over to partial algebras. This leads to a unique framework for attributed graphs, where they can be seen as *one partial algebra*. As the transformation concept we use a single pushout. Hence transformations are performed in one step. The approach was triggered by the needs of the application. Undefinedness can be modeled very naturally and can be required in the left hand side of a rule. Furthermore the complex relationships between objects need not artificially be coded into a couple of simple relationships. This avoids additional consistency conditions. Let us now introduce the basic notions from universal algebra we need for our approach.

**Definition 1 Signature, Algebra, Total Morphism.** A *signature* $SIG = (S, OP, dom : OP \to S^*, cod : OP \to S)$ consists of a set of sorts $S$, a set of operations $OP$ and two functions which assign to each operation its argument sorts and its target sort, resp. For short we write $op : w \to s_{n+1}$ if $w = s_1 ... s_n$, $dom(op) = w$ and $cod(op) = s_{n+1}$. A *partial SIG-algebra* $A$ consists of a $S$-set[3] $A_S$ of carrier sets and a family of partial mappings $A_{OP}$ such that $op^A : A_w \to A_s \in A_{OP}$ if $op : w \to s$. If all operations are defined for all elements in the domain, $A$ is a *total SIG-algebra*.

A partial SIG-algebra $B$ is a *subalgebra* of another partial algebra $A$ w.r.t. the same signature, written $B \subseteq A$ if $B_S \subseteq A_S$ and $op^B = op^A|_B$.[4] This concept implies that for each partial algebra every subset family of its carrier sets can be uniquely extended to a subalgebra.

A total morphism $f : A \to B$ between two partial SIG-algebras $A$ and $B$ is a $S$-mapping $f : A_S \to B_S$ such that $op^A$ defined for $x \in A_w$ implies $op^B$ defined for $f(x)$ and $op^B(f(x)) = f(op^A(x))$. The usual morphisms on total algebras are a special case of this notion. The category of partial algebras and total morphisms is denoted by $\underline{P - SIG}$. $\diamondsuit$

**Definition 2 Partial Morphism between Partial Algebras.** A *partial morphism* $f : A \to B$ between partial SIG-algebras $A$ and $B$ is a total morphism $f! : A(f) \to B$ from a subalgebra $A(f) \subseteq A$ to $B$. The category of all partial SIG-algebras and partial morphisms between them is denoted by $\underline{P - SIG^P}$. Composition of morphisms in $\underline{P - SIG^P}$ is componentwise composition of partial maps, which is associative. The total identities on the objects in $\underline{P - SIG}$ are also the identities in $\underline{P - SIG^P}$. A partial morphism is *closed for an operation* $op : w \to s$ if the definedness of $op^B$ on $f(a)$ implies that $op^{A(f)}$ is defined on $a \in A_w$. A partial morphism is *closed* if it is closed for all operations $op \in OP$. $\diamondsuit$

Closed morphisms do not "add definedness" for already existing arguments. Due to the notion of a subalgebra definedness can be "forgotten" by a morphism only if the worth objects of the operation are deleted.

### 3.1 Pushouts in Categories of Partial Algebras

In the following we investigate categories of the type $\underline{P - SIG^P}$ w.r.t. existence of pushout constructions which shall provide the basis for a single pushout transformation concept in these categories.

***Lemma 3 Coproducts in $\underline{P - SIG^P}$.*** *The coproduct of two partial algebras $A$ and $B$ in $\underline{P - SIG^P}$ is the (componentwise) disjoint union of $A$ and $B$, i.e., $A + B = A \uplus B$, together with the total embeddings $a : A \to A + B$ and $b : B \to A + B$ mapping $A$ and $B$ identically to their copies in $A \uplus B$.* $\diamondsuit$

*Proof.* It is easy to construct a unique morphism $u : A + B \to X$ for each pair $x : A \to X$ and $y : B \to X$ of partial morphisms such that $u \circ a = x$ and $u \circ b = y$. $\diamondsuit$

[3] The notion of $S$-sets is short for "$S$-indexed family of sets".

[4] Note that the subalgebra notion used here is a relative subalgebra in sense of [2]. The notion $op^B = op^A|_B$ denotes the domain and codomain restriction of a partial function.

Note that the coproduct construction of partial algebras coincides with the construction of coproducts in the category $\underline{SET^P}$ of sets and partial mappings if we forget the operational structure (compare [31]). The same property holds for coequalizer constructions in $\underline{P-SIG^P}$ which is demonstrated in the following. Essential for this result is the technical lemma below.

***Lemma 4 Morphism Extension of Single-Valued Partial Mappings.*** *Let $SIG = (S, OP)$ be the underlying signature of the category $\underline{P-SIG^P}$, let $A \in \underline{P-SIG^P}$, let $B$ be an $S$-indexed family of sets such that $B_s = \{*\}$ for exactly one sort $s \in S$ and $B_{s'} = \emptyset$ for all $s' \neq s$, and let $f : A \to B$ be a family of partial mappings from the carriers of $A$ to $B$, then the carriers of $B$ can be extended to a partial $SIG$-algebra $B^*$ such that $f$ becomes a $\underline{P-SIG^P}$-morphism.* $\diamondsuit$

*Proof.* Define the operational structure on $B$ as follows. For all operators $op : s \times ... \times s \to s \in OP$ let $op(*, ..., *) = *$ and let all other operators be undefined everywhere. Let $A(f)$ be the definedness area of $f$. We have pointed out above that for a partial algebra $A$, each subset family of its carrier sets uniquely induces a subalgebra of $A$. Hence $A(f)$ can be considered as a subalgebra of $A$, i.e., $A(f) \subseteq A$. Then, by construction of the operator structure on $B$, the morphism $f! : A(f) \to B$ is a total morphism on partial algebras. $\diamondsuit$

***Lemma 5 Coequalizers in $\underline{P-SIG^P}$.*** *If two morphisms $f, g : A \to B$ in $\underline{P-SIG^P}$ have a coequalizer, it coincides with the coequalizer of the underlying partial mappings of $f$ and $g$, i.e., if $U : \underline{P-SIG^P} \to \underline{SET^P}$ is the functor which forgets the operational structure in $\underline{P-SIG^P}$, $h$ is the coequalizer of $f$ and $g$ in $\underline{P-SIG^P}$ and $k$ is the coequalizer of $U(f)$ and $U(g)$ in $\underline{SET^P}$, then $U(h) \cong k$.* $\diamondsuit$

$$
\begin{tikzcd}
 & & U(C) \\
U(A) \arrow[r, shift left, "U(f)"] \arrow[r, shift right, "U(g)"'] & U(B) \arrow[ur, "U(h)"] \arrow[dr, "k"'] \arrow[r, phantom, "="] & {} \\
 & & D \arrow[uu, dashed, "x"']
\end{tikzcd}
$$

**Fig. 1.** Coequalizers in $\underline{P-SIG^P}$ and $\underline{SET^P}$

*Proof.* Consider Fig. 1 in $\underline{SET^P}$. Since $k$ is the coequalizer of $U(f)$ and $U(g)$ in $\underline{SET^P}$ and $h$ is the coequalizer of $f$ and $g$ in $\underline{P-SIG^P}$, we have $U(h) \circ U(f) = U(h \circ f) = U(h \circ g) = U(h) \circ U(g)$ and a unique partial mapping $x : D \to U(C)$ such that $x \circ k = U(h)$. Due to their coequalizer property $U(h)$ and $k$ are epimorphisms which means in the context of partial mappings and morphisms that they are surjective. We show that $U(C)$ and $D$ are isomorphic by showing $x$ being (1) total and (2) injective.

Suppose $x$ is not total which means that there is $d \in D_s$ for which $x$ is undefined. Let $k^{-1}(d)$ be all pre-images of $d$ under $k$ in $U(B)_s$. Define the following family $Y$ of sets: $Y_s = \{*\}$ and $Y_{s'} = \emptyset$ for all $s' \neq s$. Now we have the following family of partial mappings $y : U(B) \to Y$: $y_s$ is defined for all $b \in k^{-1}(d)$ by $y_s(b) = *$ and undefined otherwise. From $k \circ U(f) = k \circ U(g)$ it follows immediately that $y \circ U(f) = y \circ U(g)$. With Lemma 4 $Y$ can be extended to a $\underline{P-SIG^P}$-algebra $Y^*$ such that $y$ becomes a $\underline{P-SIG^P}$-morphism $y : B \to Y^*$. We get $y \circ f = y \circ g$ since the same property holds for the underlying mappings. With $h$ being coequalizer there is $u : C \to Y^*$ such that

$u \circ h = y$. Hence $U(h)$ is defined for all elements in $k^{-1}(d)$ leading to a contradiction to $x \circ k = U(h)$ since $x \circ k$ is undefined for all elements in $k^{-1}(d)$ due to $x$ being undefined for $d$ by assumption. Thus $x$ is total.

Now assume that $x$ is not injective, i.e., there exist $d_1, d_2 \in D_s$ with $x(d_1) = x(d_2)$. Let $k^{-1}(d_1)$ and $k^{-1}(d_2)$ be all pre-images of $d_1$ and $d_2$ under $k$ in $U(B)_s$, resp. As we have pointed out above each family of subsets of the carrier sets of an algebra induces a unique subalgebra. Hence let $Z^*$ be the subalgebra induced by $Z_s = k^{-1}(d_1) \cup k^{-1}(d_2)$ and $Z_{s'} = \emptyset$ for all $s' \neq s$. A morphism $z : B \to Z^*$ can be constructed choosing $Z^*$ as the definedness area $B(z)$ and the identity as total morphism $z! : Z^* \to Z^*$. Note that $z$ is injective. From $k \circ U(f) = k \circ U(g)$ it follows immediately that $z \circ f = z \circ g$. With $h$ being coequalizer there is $v : C \to Z^*$ such that $v \circ h = z$. This means that $U(h)$ must be injective for all elements in $Z^*$ leading to a contradiction to $x \circ k = U(h)$. Thus $x$ is injective. $\diamondsuit$

***Proposition 6 Necessary Condition for Pushouts in $\underline{P-SIG^P}$.*** *If $(D, f^* : C \to D, g^* : B \to D)$ is the pushout of $f : A \to B$ and $g : A \to C$ in $\underline{P-SIG^P}$, the underlying partial mappings of $f^*$ and $g^*$ are the pushout of the partial mappings constituting $f$ and $g$ in $\underline{SET^P}$, i.e., for the forgetful functor $U : \underline{P-SIG^P} \to \underline{SET^P}$, $(U(D), U(f^*), U(g^*))$ is the pushout of $U(f)$ and $U(g)$.* $\diamondsuit$

*Proof.* The stated property holds for coproducts and coequalizers (compare Lemmas 3 and 5) and $\underline{P-SIG^P}$ has all coproducts. Hence each pushout in $\underline{P-SIG^P}$ can be constructed in two steps, namely by first constructing the coproduct $(B + C, b : B \to B + C, c : C \to B + C)$ and then constructing the coequalizer $e : B + C \to D$ for $b \circ f$ and $c \circ g$. $\diamondsuit$

***Theorem 7 Pushouts in $\underline{P-SIG^P}$.*** *The existence of a pushout for two morphisms $f : A \to B$ and $g : A \to C$ in $\underline{P-SIG^P}$ can be characterized by the following property of the pushout $(D, f^* : U(C) \to D, g^* : U(B) \to D)$ for the underlying mappings $U(f)$ and $U(g)$ in $\underline{SET^P}$: $f$ and $g$ have a pushout if and only if (1) and (2) below define partial mappings on $D$ for each operator $op : s_1 \times ... \times s_n \to s_{n+1} \in SIG$:*

1. *If $op^B(x_1, ..., x_n) = x_{n+1}$ and $g^*$ is defined for $x_i$ with $i = 1...n+1$ define $op^D(g^*(x_1), ..., g^*(x_n)) = g^*(x_{n+1})$.*
2. *If $op^C(x_1, ..., x_n) = x_{n+1}$ and $f^*$ is defined for $x_i$ with $i = 1...n+1$ define $op^D(f^*(x_1), ..., f^*(x_n)) = f^*(x_{n+1})$.*

*The pushout in $\underline{P-SIG^P}$ is then constituted by the partial algebra $D$, where the operations are defined by (1) and (2) above, and $f^*$ and $g^*$ are homomorphic by construction.* $\diamondsuit$

*Proof.* Suppose (1) and (2) above do not define partial mappings. It implies that there is no operational structure on $D$ such that $f^*$ and $g^*$ become homomorphic. Thus, Proposition 6 guarantees that there is no pushout.

Obviously $f^* \circ f = g^* \circ f$ due to the same property of the underlying mappings. Now let there be a $\underline{P-SIG^P}$-algebra $E$ and partial morphisms $f' : C \to E$ and $g' : B \to E$ such that $g' \circ f = f' \circ g$. Then the same property holds for the underlying mappings and we obtain a unique mapping $u : D \to E$ such that $u \circ f^* = f'$ and $u \circ g^* = g'$. Hence, it remains to show that $u$ is compatible with the operations defined on $D$: Let $op : s_1 \times ... \times s_n \to s_{n+1} \in SIG$, $op^D(x_1, ..., x_n) = x_{n+1}$, and $u$ be defined for $x_i$ with $i = 1...n+1$. Definedness of $op^D$ for $(x_1, ..., x_n)$ is due to either (1) or (2) from above. So let us assume without loss of generality $x_i = f^*(c_i)$ with $i = 1...n+1$ and $op^C(c_1, ..., c_n) = c_{n+1}$. Now $u \circ f^* = f'$ implies that $f'$ is defined for $c_i$ $(i = 1...n+1)$ and its morphism property implies $op^E(f'(c_1), ..., f'(c_n)) = f'(c_{n+1})$. Substituting $u \circ f^*$ for $f'$ and $x_i$ for $f^*(c_i)$ in this equation provides $u(x_{n+1}) = u(op^D(x_1, ..., x_n)) = op^E(u(x_1), ..., u(x_n))$. $\diamondsuit$

Theorem 7 shows how pushouts in $\underline{P-SIG^P}$ can be constructed whenever they exist. In the following this leads to the observation that pushouts do not always exist on one hand and on the other hand it allows us to prove a sufficient condition for the existence of pushouts. Note that the existence of pushouts in the category $\underline{GS^P}$ of partial morphisms between total algebras with unary operations only as it was proven in [26] is also based on the underlying pushout in $\underline{SET^P}$, i.e., the inclusion functor $I : \underline{GS^P} \rightarrow \underline{P-SIG^P}$ preserves colimits.

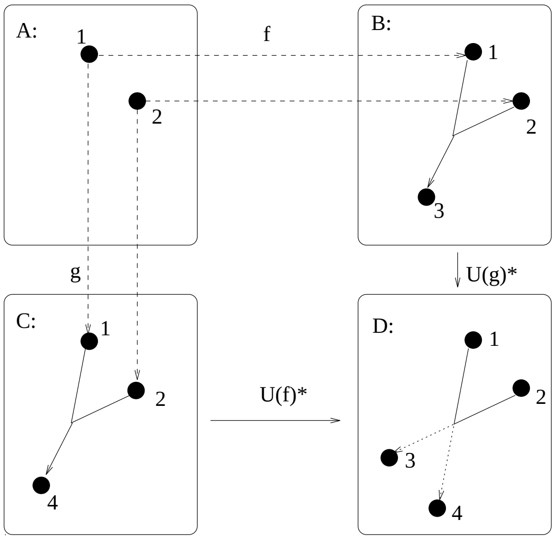


**Fig. 2.** Failure of Pushout Construction in $\underline{P-SIG^P}$

**Example 1 Non-existence of Pushouts in $\underline{P-SIG^P}$.** Consider a signature SIG with a single sort $S$ and a single binary operator *op* on this sort. Figure 2 depicts two morphisms in $\underline{P-SIG^P}$: Elements of $S$ are drawn as vertices, the assignments of *op* are drawn as solid edges and the morphism assignments are visualized by dashed edges. Both morphisms "add" definedness to the same object (1) in $A$. Hence there is no chance to define a unique operational structure on the pushout object $D$ of the underlying mappings such that $U(f^*)$ and $U(g^*)$ become homomorphic. The situation depicted in Fig. 3 shows that even if both morphisms are closed and one of them is injective the pushout may not exist. $\diamondsuit$

Non-injectivity of one morphism is enough to destroy the well-definedness of the operations of the pushout object $D$ constructed as in Theorem 7. Only if both morphisms are injective, different definitions for the operations may not overlap in the pushout object. Furthermore because of the example depicted in Fig. 2 at least one morphism has to be closed. Another possibility is that both morphisms are closed such that they may not add contradicting definedness for operations and one morphism is isomorphic for the objects. The second requirement is adequate to exclude examples like presented in Fig. 3 because the closed morphism $g$ can only add definedness for newly added objects.

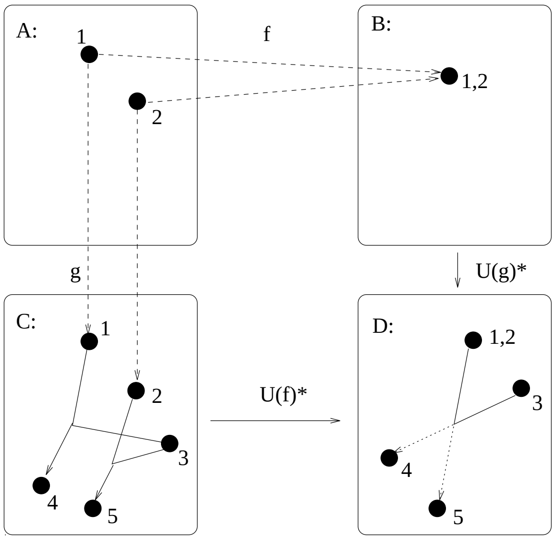


**Fig. 3.** Failure of Pushout Construction for Closed Morphisms

For the proof of this sufficient condition for the pushout existence in $\underline{P-SIG^P}$ we need some properties of the underlying pushout in $\underline{SET^P}$.

***Lemma 8 Properties of Pushouts in $\underline{SET^P}$.*** *If $(D, f^* : C \to D, g^* : B \to D)$ is the pushout of two partial mappings $f : A \to B$ and $g : A \to C$ in $\underline{SET^P}$ the following properties hold:*

1. *Pushouts preserve injective morphisms, i.e., $f^*(c_1) = f^*(c_2) \Rightarrow \exists a_1, a_2 \in A$ with $g(a_1) = c_1, g(a_2) = c_2$ and $f(a_1) = f(a_2)$.*
2. *Pushouts preserve closed morphisms.*
3. *$f, g$ total $\Rightarrow f^*, g^*$ total.*
4. *$f^*$ and $g^*$ are jointly surjective.*
5. *$f, g$ injective and $g$ total $\Rightarrow g^*$ total.*
6. *$f^*(c) = g^*(b) \Rightarrow \exists c' \in C, b' \in B$ and $a \in A$ with $f^*(c) = f^*(c'), g^*(b) = g^*(b')$ and $f(a) = b'$ and $g(a) = c'$.*

$\diamondsuit$

*Proof.* Straight forward from the construction of pushouts in $\underline{SET^P}$. $\diamondsuit$

***Theorem 9 Sufficient Condition for Pushouts in $\underline{P-SIG^P}$.***
*Two morphisms $f : A \to B$ and $g : A \to C$ have a pushout in $\underline{P-SIG^P}$ if for all operations $op : s_1 \times ... \times s_n \to s_{n+1}$ one of the following conditions holds:*

1. *$g$ is closed for $op$, and*
   *$f$ is injective for all pre-images under $g$ of arguments for which $op^C$ is defined and, vice versa,*
   *$g$ is injective for all pre-images under $f$ of arguments for which $op^B$ is defined.*
2. *$f$ and $g$ are closed for $op$ and $f$ is isomorphic for all argument sorts of $op^A$.*

$\diamondsuit$

*Proof.* We show that the pushout object $D$ of the underlying partial mappings can be enriched to a partial SIG-algebra $D^*$ (compare Theorem 7) if we define the operations as required there, i.e., we prove the well-definedness of the operations.

Suppose condition (1) is satisfied for *op*. We investigate three cases. First we assume that possible arguments of $op^D$ have more than one pre-image under $g^*$. In this case it follows from pushout property in $\underline{SET^P}$ (see Lemma 8) that they also have pre-images under $f$ in $A$ which are identified by $g$. Hence $op^B$ is undefined by condition (1).

With an analogous argument we get that $op^D$ is well-defined even if possible arguments have more than one pre-image under $f^*$. Let us now assume $op^D(f^*(x_1), ..., f^*(x_n)) = f^*(x_{n+1})$ and we have $op^D(g^*(x'_1), ..., g^*(x'_n)) = g^*(x'_{n+1})$ with $f^*(x_i) = g^*(x'_i)$ for $i = i...n$. By Lemma 8 we can choose $y_i$ and $y'_i$ with $f^*(x_i) = f^*(y_i)$ and $g^*(x'_i) = g^*(y'_i)$ such that $y_i$ and $y'_i$ for $i = 1...n$ have a common pre-image in $A$. From the first two cases considered above it follows that $x_i = y_i$ and $x'_i = y'_i$ for $i = 1...n$. Because $g$ is closed for *op*, $x_{n+1}$ has a pre-image under $g$ in $A$. Because morphisms preserve definedness it follows that this pre-image must be mapped to $x'_{n+1}$ by $f$. Pushouts commute and therefore it follows $f^*(x_{n+1}) = g^*(x'_{n+1})$.

Now suppose condition (2) is satisfied for *op*. We investigate the same three cases as above. First we assume that $op^D(g^*(x_1), ..., g^*(x_n)) = g^*(x_{n+1})$ and $op^D(g^*(x'_1), ..., g^*(x'_n)) = g^*(x'_{n+1})$ with $g^*(x_i) = g^*(x'_i)$ for $i = 1...n$. $f$ is isomorphic for all argument sorts of *op*. Hence there are pre-images in $A$ for all $x_i, x'_i$ with $i = 1...n + 1$. By Lemma 8 these pre-images of $x_i$ and $x'_i$ are identified by $g$ for $i = 1...n$. Because pushouts commute, $g$ must be defined for the pre-images of $x_{n+1}$ and $x'_{n+1}$, resp. The property of each morphism then guarantees that $g$ identifies their pre-images. Thus by the commutativity of pushouts it follows $g^*(x_{n+1}) = g^*(x'_{n+1})$.

The second case and the third case are straight forward like the second and the third case for condition (1). $\diamondsuit$

### 3.2 Interpretations of Partial Algebras as Attributed Graphs

Attributed graphs are directed graphs enriched by a data type component and attributions which assign data values to graphical objects. In this section we start with the well-known algebraic view of directed graphs and show how this view can be extended to the framework of partial algebras. For each single extension we show how it is used to model the operational behavior of TROLL *light* specifications.

Directed graphs can be seen as total algebras w.r.t. the following signature:

$$\begin{aligned} \mathrm{GRAPH} = {} & \underline{\mathrm{SORTS}}\ V, E \\ & \underline{\mathrm{OPNS}}\ \ s, t : E \to V \end{aligned}$$

This means that each graph $G$ consists of a set of vertices $G_V$, a set of edges $G_E$, and two unary mappings $s^G, t^G : G_E \to G_V$ which provide source and target vertices for each edge.

In a partial algebra w.r.t. the GRAPH signature the source or target mapping for each edge might be undefined. We call such edges dangling. But what is the interpretation of a dangling edge? Normally edges are interpreted as connections between objects. In this sense an edge with a missing source or target vertex *specifies* that there might be a connection but currently it is not established. This leads to the interpretation that dangling edges “do not exist currently”. They only specify the possible existence of edges. Typing of edges is done in the signature. So the existence or non-existence of source resp. target vertices does not affect the edge type.

More general graphical structures can be seen as algebras w.r.t. signatures which contain unary operator symbols only ([26]). Hence one can model graphs with different types of vertices and edges.

**Example 2 Basic Model of TROLL light Object Communities.** The general idea is to model objects as vertices and relationships between objects (i.e., the sub-object relationship or object-valued attributes) as edges. The following graph signature is part of the signature for our example of a TROLL *light* template collection described in Sect. 2:

$$
\begin{array}{llll}
\text{GSIG} = & \underline{\text{SORTS}} & \text{Author, Book, Library,} & \text{** vertex sorts for objects **} \\
 & & \text{Author(Book), ...} & \text{** edge sorts **} \\
 & \underline{\text{OPNS}} & \text{s: Author(Book)} \rightarrow \text{Book} & \\
 & & \text{t: Author(Book)} \rightarrow \text{Author} &
\end{array}
$$

For the three templates we have three sorts of vertices. For the object-valued attribute Author of template Book we have an edge sort. ♢

Data types can be specified algebraically [11, 8, 12, 37]. In this context we presuppose such an algebraic view of the desired data types, i.e., a signature.

**Example 3 Data Types in TROLL light Object Communities.** Our example specification contains three data types, which we assume to be specified somewhere else. The data signature is:

$$\text{DSIG} = \text{STRING} + \text{DATE} + \text{NAT}$$

♢

The signature for an attributed graph consists of the signature for the graph, the signature for the data types and the attributions, which assign data values to graphical objects. Attributions can be seen as a special kind of edges.

**Example 4 Extension to Model Data-valued Attributes.** We already saw above that object-valued attributes in TROLL *light* are modeled by edges. Attributions in the sense of attributed graphs are used to model data-valued attributes as for example the attribute Title of template Book. A part of the attributed graph signature for our template collection in Sect. 2 containing data valued attributes is:

$$
\begin{array}{lll}
\text{ATTRGRA} = \text{GSIG} + \text{DSIG} + & & \\
 & \underline{\text{SORTS}} & \text{Title(Book)} \\
 & \underline{\text{OPNS}} & \text{s: Title(Book)} \rightarrow \text{Book} \\
 & & \text{t: Title(Book)} \rightarrow \text{String}
\end{array}
$$

♢

By now we only modeled (1:1)-relationships between objects and objects or between objects and data elements. But within TROLL *light* specifications we also have (1:$n$)-relationships, where each of the $n$ objects or data elements (being in relationship with the same object) is uniquely identified by one or more parameters. If these relationships shall be modeled using edges we need the notion of a *family of outgoing edges* of a vertex where each single edge can be uniquely identified, using so-called *indices*.

If we want to model graph like structures as total algebras it is essential that the signatures contain unary operator symbols only. But within the framework of partial algebras all types of operations are allowed. This is quite useful for the specification of families of edges.

All edges within the same family of outgoing edges have the same source vertex. Hence we use a unary source operation to assign the source vertex to the family. The target vertices may be different and depend on the indices. Algebraically this is represented by a non-unary target operation mapping the edge family and all indices to the target vertex.

Above we stated that dangling edges are only specifications of edges. This certainly is also true for edge families, i.e., an edge within a family only exists if its source and target operation are defined.

**Example 5 Extension of Model to Parameterized Sub-object Relationship.** To include the sub-object relationship Books of template Library we extend our signature as follows:

EXTATTRGRA = ATTRGRA +
  SORTS  Books(Library)
  OPNS  s: Books(Library) → Library
     t: Books(Library), Author, String → Book

$\diamondsuit$

Now we summarize the above considerations in the definition of an attributed graph.

**Definition 10 Attributed Graph.** An *attributed graph signature* $AGSIG = (GS, DS, OP)$ consists of a set of *graphical sorts* $GS$, a set of *data sorts* $DS$ and a set of operations $op : w \to s$ which is the disjoint union of the following sets: (1) The set of *graphical* or edge operations with $w \in GS^*, s \in GS$. (2) The set of *data* or computational operations with $w \in DS^*, s \in DS$. (3) *Attributions* map graphical objects to data objects, i.e., $w \in GS^*, s \in DS$. (4) *Indexed attributions* are operations with $w \in (GS \cup DS)^*, s \in DS$ and (5) for *indexed edges* we have $w \in (GS \cup DS)^*, s \in GS$.

An *attributed graph* is a partial algebra w.r.t. an attributed graph signature. The *indices* of an attributed graph $G$ build a family of sets $(I_s)_{s \in S}$, where $s$ is a data sort and $i \in I_s$, if there exists an indexed edge or an indexed attribution $op : s_1 \times ... \times s_n \to s_{n+1}$ with $s_j = s$ for $j \in \{1..n\}$ and $op^G(x_1, .., x_n)$ is defined and $x_j = i$ holds. $\diamondsuit$

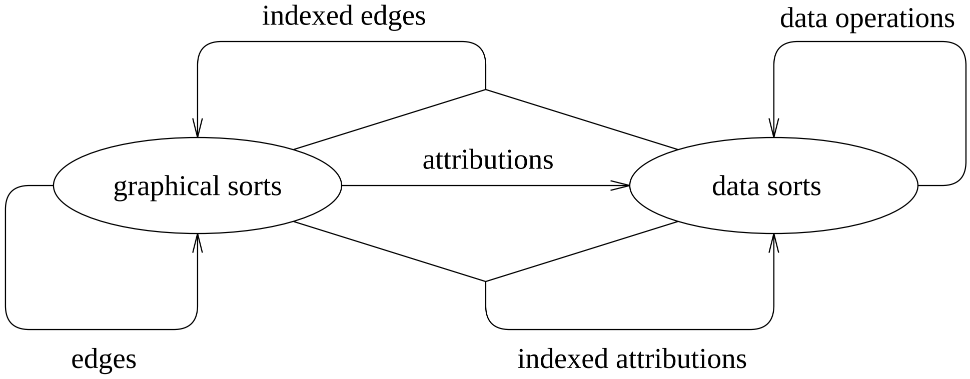


**Fig. 4.** Schema of Attributed Graph Signatures

Note that attributed graphs contain no operations mapping only data sorts to graphical ones. Figure 4 visualizes the schema of attributed graph signatures. Indices build no special sorts. In a given graph it depends on the definedness of the operations (indexed attributions and indexed edges) which data elements are used as indices and which not.

### 3.3 Graph Transformations

Theorem 9 provides us with two sufficient conditions for the existence of pushouts in $\underline{P-SIG^P}$. First, a rule would be applicable at any redex if we restrict rules to injective morphisms and redices to closed injective morphisms. This might be adequate if one thinks of the graphical part, but for the data part injective redices are unsatisfactory. The data part of a rule certainly contains variables. If they are not be identified by the redex, this would increase the number of rules in a not acceptable way.

Second, a rule would be applicable at any redex if we restrict both rules and redices to closed morphisms and one of them to an isomorphism. For the graphical part an isomorphism is neither possible as a redex nor as a rule. But for the data part rules may be isomorphisms, because the data part is assumed to be specified somewhere else and only used here without any change. This approach is already used for total attributed graphs (see [27]).

What we do is to combine both conditions. Rules and redices are partial morphisms between partial algebras. But if we restrict these morphisms to the data or graphical part of the attributed graph, respectively, they satisfy different conditions. Rules for the graphical part are injective, rules for the data part isomorphisms. Redices are total, closed morphisms, which are injective for the graphical part.

The indices play a special role. They are used to uniquely identify edges in a family or attributions. If they are identified by a redex the uniqueness of the identification would be lost. Hence redices should be injective for indices.

**Definition 11 Rule, Redex, Derivation.** A *rewriting rule* $r : L \to R$ is an injective partial morphism in $\underline{P-SIG^P}$, which is isomorphic for all data sorts. The *indices* of the rule $r$ are defined by the union of the indices of $L$ and $r^{-1}$ applied to the indices of $R$. A *redex* for $r$ in an object $G$ is a total, closed morphism $m : L \to G$, which is injective for all graphical sorts and all indices used in $r$. The *direct derivation* of $G$ with the rule $r$ at a redex $m$ is the pushout of $m$ and $r$ in $\underline{P-SIG^P}$. The corresponding pushout object $H$ is called *derived graph.* A direct derivation from $G$ to $H$ with a rule $r$ is denoted by $\overset{r}{\Longrightarrow}$. ♢

The idea of a rule is to describe as universally as possible the desired behavior of a transformation. For the data type part of the left and right hand side of the rule this means that one will choose a syntactical algebra, i.e., the *total term algebra with variables* over the subsignature containing only data sorts and data operations. The variables are normally used to store the actual value of an attribute during the transformation. Note that the injectivity of redices for indices means that there are variables in the left hand side of the rule which have to be evaluated to different values.

Due to the notion of a redex as a closed morphism our framework includes some kind of *negative application condition* for a rule. If an operation is undefined for arguments in the left hand side of the rule, the rule is not applicable to a graph if it is defined there. This is a very strong requirement which has to be considered in any application.

We divide between optional relationships (modeled by edges) and inevitable known relationships (modeled by operations). Note that this may be problematic if operations in the data type component are used in the classical way. Variable instantiations by a redex may be not possible due to the existence of additional structure in the graph the rule shall be applied to.

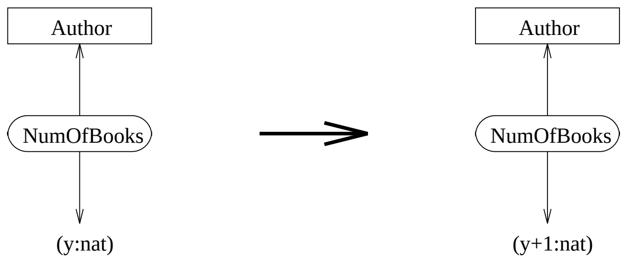


**Fig. 5.** Example for a Valuation Rule

**Example 6 Valuation Rule for Event `newBook`.** The valuation graph grammar rule for event `newBook` depicted in Fig. 5 removes the old attribution and inserts the new attribution in accordance with the TROLL *light* valuation formula. $\diamondsuit$

***Proposition 12 Direct Derivation.*** *The direct derivation of a graph $G$ with a rule $r : L \to R$ at a redex $m : L \to G$ always exists.* $\diamondsuit$

*Proof.* We show that the pushout object $D$ of the underlying partial mappings can be enriched to a partial SIG-algebra $D^*$ (compare Theorem 7) using the sufficient conditions from Theorem 9, i.e., for all operations of the attributed graph signature one of the stated conditions holds.

Redices are closed morphisms and rules are injective. Hence the conditions reduce to (1) $m$ is injective for all pre-images under $r$ of arguments for which $op^R$ is defined and (2) $r$ is closed for $op$ and isomorphic for all argument sorts of $op^L$.

If the operation is a graphical one or an attribution condition, condition (1) holds because rule and redex are injective for all (graphical) argument sorts. For all data operations condition, condition (2) is satisfied because the rule is isomorphic for all (data) argument sorts and closed for all data operations because it is isomorphic for the value sort (a data sort) too. Indexed edges and indexed attributions fulfill condition (1) as far as the redex $m$ is injective for all indices. $\diamondsuit$

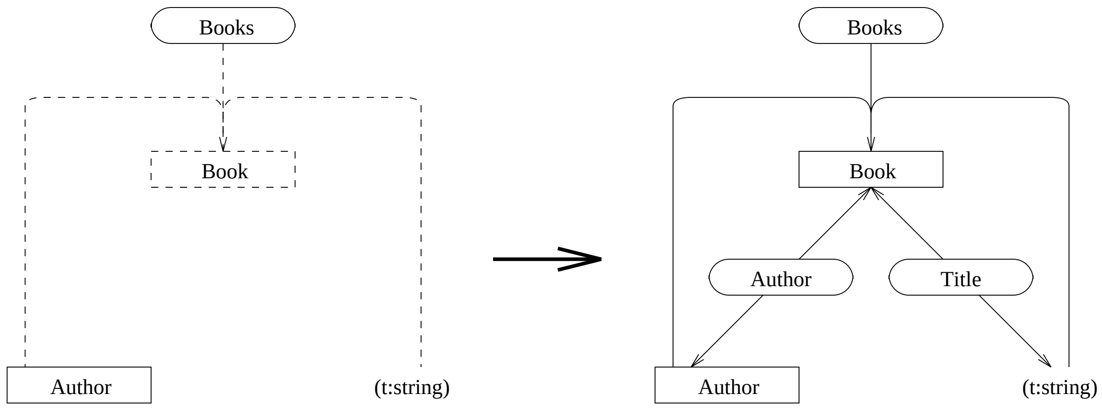


**Fig. 6.** Example for a Valuation Rule

**Example 7 Interaction Rule for Event `newBook`.** The interaction graph grammar rule in Fig. 6 for event `newBook` inserts a new `Book` node only if there is no `Book` node with the connections depicted by dashed lines. $\diamondsuit$

## 4 Synchronization via Amalgamation

The basic concept of amalgamation allows to synchronize two rules over a common sub-rule, which describes the effects the application of both rules should have. The effect of the synchronized application of both rules is reached by first constructing a new rule (the so-called amalgamated rule) and then by applying it. The fundamental result in this context is that the necessary synchronization can be minimized, i.e., each direct derivation with the amalgamated rule can be simulated by first applying the common sub-rule and then (locally) remainders of both rules.

**Definition 13 Sub-rule.** A rule $r1 : L1 \to R1$ is a *sub-rule* of a rule $r2 : L2 \to R2$ if

1. there are total, closed, injective morphisms $i : L1 \to L2$ and $j : R1 \to R2$ such that $r2 \circ i = j \circ r1$ and
2. the universal morphism $(r2 - r1)$ from the pushout object $R$ of $r1$ and $i$ to $R2$ is a rule.[5] We call $(r2 - r1)$ the remainder in the following.

$\diamondsuit$

A rule $r1$ is called subrule of a rule $r2$ if $r2$ subsumes all the effects of $r1$ and adds some more. The properties of the subrule embeddings $i$ and $j$ make sure that this becomes true for deletion and gluing. Requirement 2 is needed for the addition of new objects (adding a new object is no subeffect of preserving an object).

$$\begin{tikzcd}
L1 \arrow[r, "r1"] \arrow[d, "i"'] \arrow[dr, phantom, "="] & R1 \arrow[d, "j"] \\
L2 \arrow[r, "r2"] \arrow[d, "m"'] & R2 \\
G &
\end{tikzcd}$$

**Fig. 7.** Induced Redex for Sub-rule

The fact that redices for rules induce redices for their sub-rules is important for the minimization of the synchronization. Instead of the rule first its sub-rule should be applied.

***Proposition 14 Redex for a Sub-rule.*** *If $m : L2 \to G$ is a redex for a rule $r2 : L2 \to R2$ and $r1 : L1 \to R1$ is a sub-rule of $r2$ as depicted in Fig. 7, then $m \circ i$ is a*

[5] The pushout always exist due to Theorem 9.

*redex for $r1$.* ♢

*Proof.* Total closed injective morphisms are closed under composition. It remains to show that $m \circ i$ is injective for the indices of $r1$. With $Ind(L1)$ being the family of sets of indices of graph $L1$, $i(Ind(L1))$ are indices of graph $L2$ because $i$ is total and closed. Hence is $m$ is injective for $i(Ind(L1))$. ♢

**Definition 15 Amalgamated Rule.** Let $r1 : L1 \to R1$ and $r2 : L2 \to R2$ with a common sub-rule $r : L \to R$ be given. The *amalgamated rule* $(r1 +_r r2)$ of $r1$ and $r2$ is constructed in Fig. 8. $L3$ is the pushout of $i1$ and $i2$, $R3$ is the pushout of $j1$ and $j2$ and $(r1 +_r r2)$ is the universal morphism such that $(r1 +_r r2) \circ i2^* = j2^* \circ r1$ and $(r1 +_r r2) \circ i1^* = j1^* \circ r2$. ♢

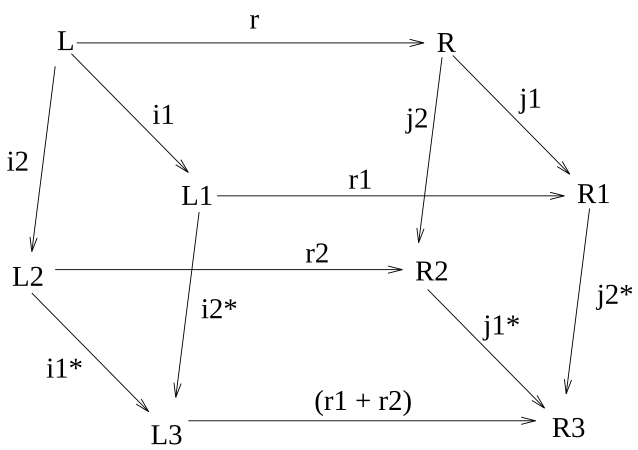


**Fig. 8.** Construction of the Amalgamated Rule

***Proposition 16 Existence of Amalgamated Rule.*** *The amalgamated $(r1 +_r r2)$ rule of two rules $r1 : L1 \to R1$ and $r2 : L2 \to R2$ with a common sub-rule $r : L \to R$ always exists.* ♢

*Proof.* Due to Theorem 9 the pushouts of $i1$ and $i2$ and of $j1$ and $j2$, respectively, always exist, because $i1, i2, j1$ and $j2$ are injective and closed. Hence there is a universal morphism $(r1 +_r r2) : L3 \to R3$. To show that $(r1 +_r r2)$ is a rule we first show its injectivity and then its surjectivity for all data sorts. Assume that $(r1 +_r r2)(x_1) = (r1 +_r r2)(x_2)$ and $x_1 \neq x_2$. By pushout property (4) for the underlying pushout in $\underline{SET^P}$ $i1^*$ and $i2^*$ are jointly surjective. Because of the injectivity of $j1^* \circ r2$ and $j2^* \circ r1$, $x_1$ and $x_2$ cannot have pre-images under the same morphism. So we assume a pre-image for $x_1$ in $L1$ and for $x_2$ in $L2$. Because of the property of the universal morphism $(r1 +_r r2)$, $j2^* \circ r1(i2^{*-1}(x_1)) = j1^* \circ r2(i1^{*-1}(x_2))$ holds. With pushout property (6) we get a common pre-image for $r1(i2^{*-1}(x_1))$ and $r2(i1^{*-1}(x_2))$ in $R$. Because of the injectivity of the remainder $(r1 - r)$ and pushout property (6) $i2^{*-1}(x_1))$ has a pre-image under $i1$ in $L$. $r$ is a sub-rule of $r2$ and hence the pre-image of $i2^{*-1}(x_1))$ under $i1$ is also a pre-image of $i1^{*-1}(x_2)$ under $i2$. From the commutativity of pushouts it follows that $x_1 = x_2$. Hence $(r1 +_r r2)$ is injective. The proof of surjectivity for the data sorts is straight forward using the surjectivity of $r1$ and $r2$ for data sorts and the fact that $j1^*$ and $j2^*$ are jointly surjective by Lemma 8 ♢

*Remark Iterated Amalgamation.* Due to the commutativity of colimits and the fact that pushouts preserve total, closed, injective morphisms the process of amalgamation can be iterated and still has a unique result. This will be demonstrated in detail by a more involved example in the next section (Fig. 15). ♢

The rest of this section is dedicated to the proof of the decomposition theorem of transformations with the amalgamated rule. The following technical proposition is necessary due to the fact that not every decomposition of a rule preserves redices.

***Proposition 17 Redex Preserving Rule Decomposition.*** *Let $r : L \to R$ be a sub-rule of $r1 : L1 \to R1$ as depicted in Fig. 9 and $m : L1 \to G$ be a redex for $r1$. Then $m$ is a redex for $r^*$ and the morphism $m^*$ induced by the direct derivation of $G$ with $r^*$ at $m$ is a redex for the remainder $(r1 - r)$.* ♢

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "i"'] \arrow[dr, phantom, "\text{P.O.}"] & R \arrow[d, "i^*"] \arrow[dr, "j"] & \\
L1 \arrow[r, "r^*"] \arrow[d, "m"'] \arrow[rr, bend right, "r1"'] \arrow[dr, phantom, "\text{P.O.}"] & X \arrow[r, "(r1-r)"] \arrow[d, "m^*"] & R1 \\
G \arrow[r] & G1 &
\end{tikzcd}$$

**Fig. 9.** Sub-rule and Remainder Decomposition

*Proof.* $m^*$ is closed by Lemma 8. $m^*$ is total for graphical sorts by pushout property (5) and for data sorts by pushout property (3) for the underlying pushout in $\underline{SET^P}$. (see Lemma 8). The injectivity for graphical sorts follows from the injectivity of $m$ for graphical sorts. It remains to show that $m^*$ is injective for the indices of the remainder $(r1 - r)$. If $m^*(x_1) = m^*(x_2)$ then $m$ must identify $r^{*-1}(x_1)$, and $r^{*-1}(x_2)$. Hence $r1(r^{*-1}(x_1)) = (r1 - r)(x_1)$ and $r1(r^{*-1}(x_2)) = (r1 - r)(x_2)$ cannot be indices of $L1$ and $R1$, resp. Assume they are indices of $X$, i.e., $op^X(y_1, ..., y_n)$ defined with $y_i = x1$ and $y_j = x_2$ for $i, j \in \{1...n\}$. By Theorem 7 definedness of $op^X$ always causes definedness of $op^{L1}$ or $op^R$. The definedness of $op^{L1}$ cannot be the reason for the definedness of $op^X$ because $r^{*-1}(x_1)$ and $r^{*-1}(x_2)$ cannot be indices of $L1$. With an analogous argument for $op^R$ making use of $(r1 - r) \circ i^* = j$ and the closedness of $j$ we get that the definedness of $op^R$ cannot be the reason for the definedness of $op^X$ leading to a contradiction. ♢

***Lemma 18 Pushout Cube.*** *If the top, bottom, front, back and the left side of a cube like the one depicted in Fig. 10 are pushouts in $\underline{P - SIG^P}$, then also the right side is a pushout.* ♢

*Proof.* The corresponding property for the underlying diagram in $\underline{SET^P}$ can be shown using pushout property (4) of Lemma 8. With Theorem 7 we get that the right side of the cube is a pushout in $\underline{P - SIG^P}$. ♢

***Theorem 19 Transformation with Amalgamated Rule.*** *As pictured in Fig. 11 Every direct derivation with an amalgamated rule $(r1 +_r r2)$ can be simulated by direct derivations with the sub-rule $r$ and the remainders $(r1 - r)$ and $(r2 - r)$.* ♢

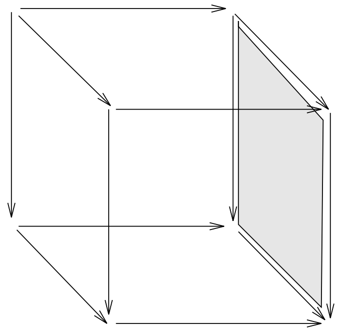

**Fig. 10.** Pushout Cube

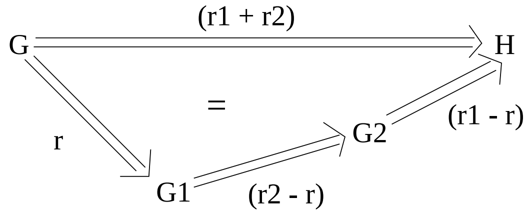


**Fig. 11.** Decomposition of Transformation with Amalgamated Rule

*Proof.* The pushout constructions of the proof are pictured in Figs. 12, 13, and 14, resp. (1) and (2) are pushouts due to the construction of the amalgamated rule $(r1 +_r r2)$. (3) and (4) are pushouts by the construction of the remainders $(r1 - r)$ and $(r2 - r)$. (4) + (5) resp. (3) + (6) is the pushout of $r$ and $i2^* \circ i1 = i1^* \circ i2$, which can be decomposed due to the existence of pushouts (5) and (6). By Lemma 18 (7) is a pushout.

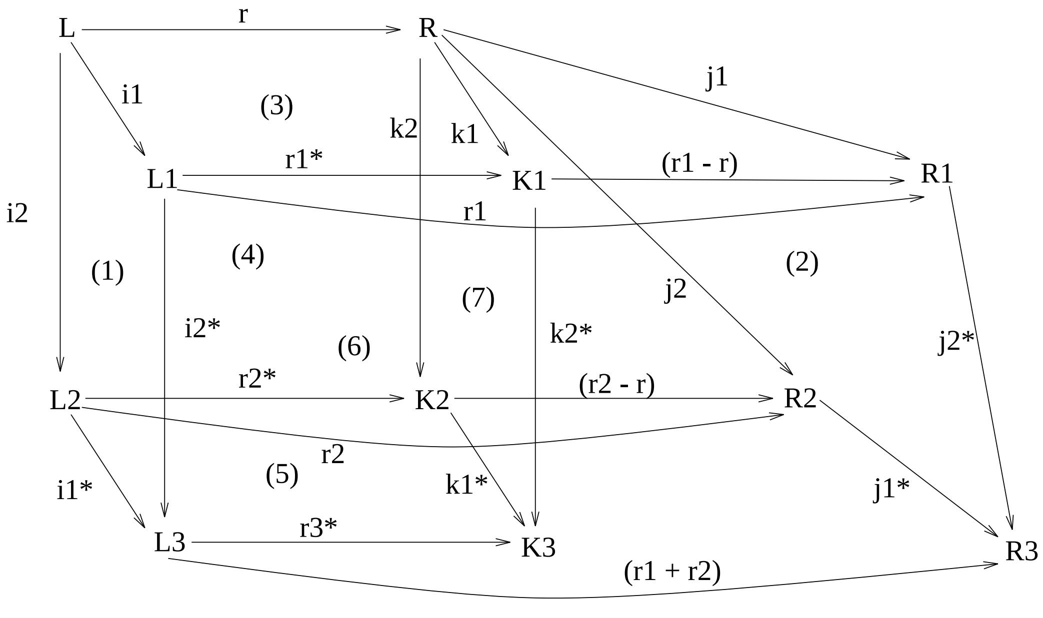


**Fig. 12.** Decomposition of the Amalgamated Rule I

(8) and (9) are constructed as pushouts of $k1^*$ and $(r2 - r)$ resp. $k2^*$ and $(r1 - r)$. $u$ and $v$ are the universal morphisms with $u \circ n1 \circ r3^* = (r1 +_r r2)$ and $u \circ n2 = j1^*$ resp. $v \circ o1 \circ r3^* = (r1 +_r r2)$ and $v \circ o2 = j2^*$. Both $u \circ n1$ and $v \circ o1$ fulfill the property of the universal morphism from $K3$ to $R3$ for pushout (7) and hence must be equal.

From this we can conclude that there is another commutative diagram (10) with $u \circ n1 \circ k2^* = j2^* \circ (r1 - r)$. To show that (10) is a pushout, i.e., $(R3, u, j2^*)$ is pushout of $(r1 - r)$ and $n1 \circ k2^*$, we assume that there exists $(X, x1, x2)$ with $x1 \circ n1 \circ k2^* = x2 \circ (r1 - r)$. Because of $x1 \circ n2 \circ j2 = x1 \circ n2 \circ (r2 - r) \circ k2 = x1 \circ n1 \circ k1^* \circ k2 = x1 \circ n1 \circ k2^* \circ k1 = x2 \circ (r1 - r) \circ k1 = x2 \circ j1$ there exists a universal morphism $z : R3 \to X$ for pushout (2) with $z \circ j2^* = x2$ and $z \circ j1^* = x1 \circ n2$. $z$ also fulfills $z \circ u = x1$, what can be proved by $z \circ u \circ n2 = z \circ j1^* = x1 \circ n2$ and $z \circ u \circ n1 \circ k2^* = z \circ j2^* \circ (r1 - r) = x2 \circ (r1 - r) = x1 \circ n1 \circ k2^*$ (sufficient because (7) + (8) is pushout), and hence serves as the universal morphism for the desired pushout. Its uniqueness follows from the uniqueness of $z$.

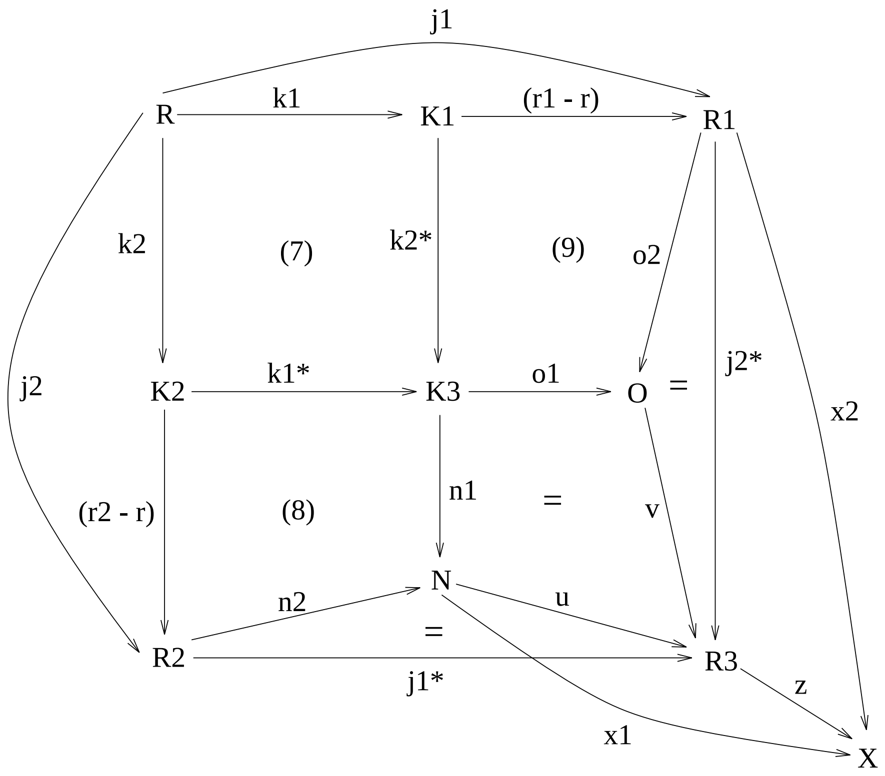


**Fig. 13.** Decomposition of the Amalgamated Rule II

Now assume that $m : L3 \to G$ is a redex for the amalgamated rule $(r1 +_r r2)$. Then by Proposition 14 $m \circ i2^* \circ i1$ is a redex for the sub-rule $r$ of $(r1 +_r r2)$. By proposition 17 $m$ is a redex for $r3^*$. (11) is the corresponding direct derivation. $(3) + (6) + (11)$ is the direct derivation of $G$ with the sub-rule $r$.

By Proposition 17 $m1^*$ is a redex for $u \circ n1$ and hence for $n1$. (12) is the corresponding direct derivation. By Proposition 14 $m1^* \circ k1^*$ is a redex for $(r2 - r)$. $(8) + (12)$ is the corresponding direct derivation.

By Proposition 17 $m2^*$ is a redex for $u$. (13) is the corresponding direct derivation. For $(10) + (13)$ being the direct derivation of $G2$ with $(r1 - r)$ it remains to show that $m2^* \circ n1 \circ k2^*$ is a redex. With Proposition 17 this reduces to $n1 \circ k2^*$ being total, closed and injective. The injectivity follows from the injectivity of $n1$ and $k2^*$.

Assume $n1 \circ k2^*$ not being total. Because $k2^*$ is total, $n1$ cannot be total for the image of $K1$ under $k2^*$. By pushout property (5) (see Lemma 8) for pushout (8), if $n1$ is not defined for some $k \in K3$, $k$ has a pre-image under $k1^*$ for which $(r2 - r)$ is not defined. With pushout property (6) $k$ has a pre-image under $k1^* \circ k2$ in $R$. $j2 \circ (r2 - r) \circ k2$

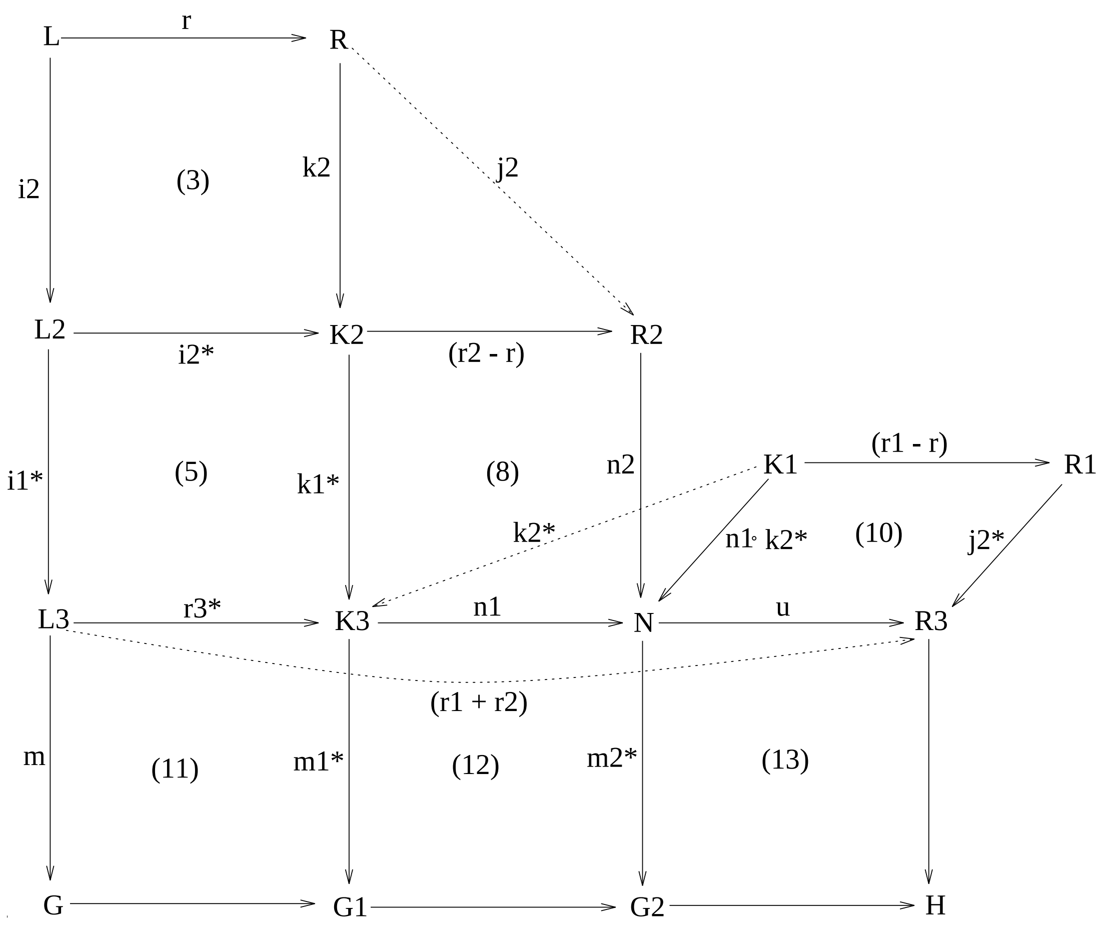


**Fig. 14.** Decomposition of an Application of the Amalgamated Rule

being total leads to a contradiction. Hence $n1 \circ k2^*$ is total.

Assume $n1 \circ k2^*$ not being closed, i.e., $op^N(n1 \circ k2^*(x_1), ..., n1 \circ k2^*(x_n)) = n1 \circ k2^*(x_{n+1})$, but $op^{K1}(x_1, ..., x_n)$ undefined. By Theorem 7 $n1(x_i)$ $i \in \{1...n+1\}$ have pre-images $y_i$, $i \in \{1...n+1\}$ under $n2$ for which $op^{R2}$ is defined. Pushout property (6) for (7) + (8) guarantees that $x_i$, $i \in \{1...n+1\}$ have pre-images under $k1$ which are also the pre-images of $y_i$, $i \in \{1...n+1\}$ under $(r2 - r) \circ k2 = j2$. $j2$ is closed and hence $op^R$ must be defined for the pre-images of $x_i$ under $k1$ leading to a contradiction with $op^{K1}(x_1, ..., x_n)$ being undefined.

Hence the derivation of $G$ with the sub-rule $r$ (pushout $(3) + (6) + (11)$) followed by the derivation with the remainder $(r2 - r)$ (pushout $(8) + (12)$) and the derivation with the remainder $(r1 - r)$ (pushout $(10) + (13)$) leads to the resulting graph $H$ which is also the direct derivation of $G$ with the amalgamated rule $(r1 +_r r2)$ (pushout $(11) + (12) + (13)$). $\diamondsuit$

## 5 Applications to TROLL *light*

We now turn to the application of the above results in order to demonstrate how they can be employed for defining the evolution of TROLL *light* object communities. Before considering the amalgamated rule for one particular example, namely the closed event set which is induced by the occurrence of the event `newBook` belonging to template `Library`, we repeat the respective parts of the above given templates.

```
TEMPLATE Library
  VALUATION   [newBook] NumberOfBooks=NumberOfBooks+1;
  INTERACTION newBook(A,T) >> Books(A,T).create(A,T),
                              A.addBook;
...
TEMPLATE Book
  VALUATION  [create(A,T)] Author=A, Title=T;
...
TEMPLATE Author
  VALUATION  [addBook ] NumOfBooks=NumOfBooks+1;
...
```

These TROLL *light* valuation and interaction rules are translated to graph grammar rules. The transition corresponding to the respective closed event set increments two attributes and inserts a new `Book` object.

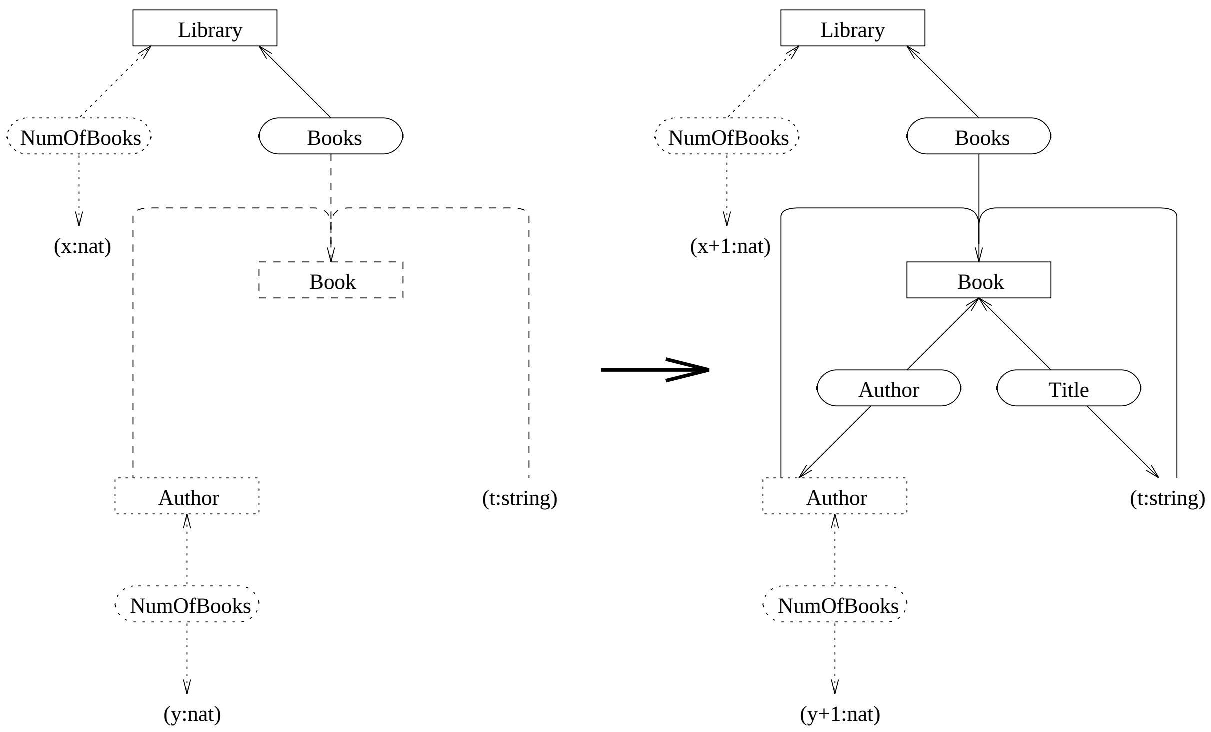


**Fig. 15.** Complex Rule Achieved via Amalgamation

The amalgamation process now combines the basic rules for the valuation and interaction parts into one complex rule as depicted in Fig. 15. In principle the rule whose left hand side consists of all non-dashed parts of the graph combines three different local rules and the interaction rule:

1. We have a valuation rule for template `Library` which increments the attribute `NumberOfBooks` (abbreviated by `NumOfBooks`). This corresponds to upper dotted part of the above diagram connecting the attribute node `NumOfBooks` with the template node `Library` and the data type variable `x:nat`.
2. Then there is another valuation rule this time for template `Author` analogously incrementing the attribute `NumOfBooks` depicted in the dotted lower part.

3. The interaction rule consists of the three template nodes Library, Author and the (added) node Book.
4. Furthermore we have a third valuation rule for template `Book` defining the context for the inserted node. On the left hand side of the rule we have an applicability condition pictured by the dashed part: The rule can be applied only if there is no template node for `Book` connected to the other nodes in the displayed manner. After application of the rule new nodes appear in the result graph as given on the right hand side of the rule.

These four rules are amalgamated using so called "trivial rules" as common subrules of the interaction rule and the different valuation rules. Trivial rules consist only of the template nodes, for instance the interface between interaction rule and valuation rule for the template `Library` of the Library node. Thus, we employ amalgamation techniques not only to describe the effect of valuation rules but also — and this is a central concern — for interaction rules.

Let us also give an example of a graph derivation employing the above described production for the closed event set induced by the event `newBook`. On the left hand side of the diagram one attributed graph representing a simple state of the object community induced by template `Library` is depicted (Fig. 16).

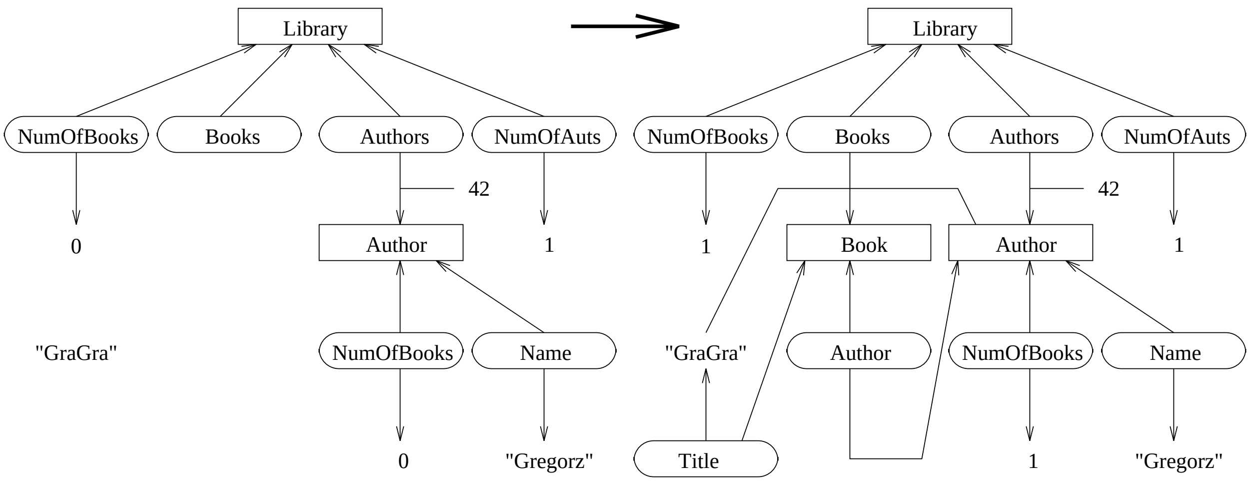


**Fig. 16.** Sample Derivation

The described transition corresponds to the closed event set with the three events `newBook(A,T)`, `Books(A,T).create(A,T)`, and `A.addBook`. In principle, the transition introduces a new object (`Book` node), modifies attributes, and establishes the connection to existing nodes.

Note that the rather complex notion of a redex in the approach introduced in this paper fits perfectly the TROLL *light* model. Injectivity for indices is used to uniquely identify subobjects and attributes. Injectivity for graphical objects solves the conflict of unallowed cyclic interaction due to the stepwise construction of the rule for a closed event set. Furthermore the closedness of a redex provides a kind of implicite negative application condition which is used, for example, to make sure that an existing object is not created again.

All these facts together should motivate the new approach. A model of TROLL *light* within the classical one based on total algebras would have to deal with the same problems and hence forces a notion of a redex of the same complexity. This should be possible to achieve using explicite negative application conditions (see [17]). But up to our discussions by now they are incompatible with the theory of amalgamation which provides us with the possibility of stepwise rule construction.

Due to space limitations we have given only a very specific example for a transition. In generally modelling the evolution of a system, the basic rules are never applied in isolation. Only amalgamated rules corresponding to closed event sets are applied and cause the state transitions.

## 6 Conclusions and Future Work

We have generalized the single pushout approach to graph grammars in order to combine it with partial algebras and partial morphisms. Our results provided a uniform framework for the transformation of attributed graphs where graphical and data parts were completely integrated. We were able to obtain attractive amalgamation results for graph derivations as in the traditional case. In particular, the notion of a redex as a closed morphism led in our framework to some kind of negative application condition for a direct derivation. The results obtained were employed for the definition of the operational semantics of the object specification languages TROLL *light*. Especially, attributed partial graphs were adequate for modeling TROLL *light* object community states because parametrized sub-object relationships were modeled naturally as families of edges and necessary negative application conditions of rules could be achieved without extending the formalism.

Nevertheless, not all features of object specification languages were considered here. For example, we did not treat the `BEHAVIOR` part of TROLL *light* templates, but our approach is powerful enough to cover this aspect of object specifications as well. A closer look has to be spend on other TROLL *light* language features like `CONSTRAINTS` and `DERIVED` attributes. Apart from treating TROLL *light* it would be interesting to draw the attention to other similar languages, in particular CMSL [36] seems to be a stimulating candidate for such considerations.


## Acknowledgements

The critical remarks of the referees have improved the quality of the paper. Thanks also to H. Ehrig for promoting our approach. Part of this work was done within the KORSO project where partners and collegues have contributed with fruitful discussions.


## References


1. M. Broy and M. Wirsing. Partial abstract data types. *Acta Informatica*, 18(1):47–64, 1982.
2. P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*, volume 32 of *Mathematical Research — Mathematische Forschung*. Akademie-Verlag, Berlin, 1986.
3. I. Claßen, M. Große–Rhode, and U. Wolter. Categorical concepts for parameterized partial specifications. Technical Report 92–42, Technische Universität Berlin, 1992.
4. S. Conrad, M. Gogolla, and R. Herzig. TROLL light: A core language for specifying objects. Technical Report 92-02, Technical University of Braunschweig, 1992.

5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, and M. Löwe. Graph grammars and logic programming. In Ehrig et al. [10], pages 221–237. Lecture Notes in Computer Science 532.

6. B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 193–242. North Holland, Amsterdam, 1990.

7. E. Dubois, P. Du Bois, and M. Petit. O-O requirements analysis: An agent perspektive. In O.M. Nierstrasz, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'93)*, pages 458–481, Berlin, 1993. Springer, LNCS 707.

8. H. D. Ehrich, M. Gogolla, and U. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Leitfäden und Monographien der Informatik. B. G. Teubner, Stuttgart, 1989.

9. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *1st Graph Grammar Workshop, Lecture Notes in Computer Science 73*, pages 1–69. Springer Verlag, 1979.

10. H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *4th International Workshop on Graph Grammars and Their Application to Computer Science.* Springer Verlag, 1991. Lecture Notes in Computer Science 532.

11. H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications 1: Equations and initial semantics*, volume 6 of *EACTS Monographs on Theoretical Computer Science.* Springer Verlag, Berlin, 1985.

12. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science.* Springer, Berlin, 1990.

13. G. Engels, R. Gall, M. Nagl, and W. Schäfer. Software specification using graph grammars. *Computing*, 31:317–346, 1983.

14. G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schür. Building integrated software development environments - part I: Tool specification. *ACM Trans. on Software Engeneering and Methodology*, 1(2):135–167, 1992.

15. P. Gabriel. The object-based specification language $\Pi$: Concepts, syntax, and semantics. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification - Proc. 8th Workshop on Specification of Abstract Data Types*, pages 254 –270, Berlin, 1993. Springer, LNCS 655.

16. M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL light. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, pages 17–32. Springer, LNCS 722, 1993.

17. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. in this volume, 1994.

18. R. Herzig, S. Conrad, and M. Gogolla. Compositional description of object communities with TROLL light. In C. Christment, editor, *Proc. Basque Int. Workshop on Information Technology (BIWIT'94): Information Systems Design and Hypermedia*, pages 183–194, Toulouse, 1994. Cépadués-Éditions.

19. R. Hull and R. King. Semantic database modelling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

20. D. Jansens and G. Rozenberg. Structured transformations and computation graphs for actor grammars. In Ehrig et al. [10], pages 446–460. Lecture Notes in Computer Science 532.

21. D. Janssens, M. Lens, and G. Rozenberg. Computation graphs for actor grammars. *Journal of Computer and System Science*, 46:60–90, 1993.

22. D. Janssens and G. Rozenberg. Actor grammars. *Mathematical Systems Theory*, 22:75–107, 1989.

23. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The TROLL language. Technical Report 91-04, TU Braunschweig, 1991.

24. S.M. Kaplan, J.P. Loyall, and S.K. Goering. Specifying concurrent languages and systems with $\Delta$-grammars. In Ehrig et al. [10], pages 475–489. Lecture Notes in Computer Science 532.

25. R. Kennaway. On "On graph rewriting". *Theoretical Computer Science*, 52:37–58, 1987.

26. M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

27. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

28. U. Montanari and F. Rossi. True concurrency in constraint logic programming. In *Proc. Int. Logic Programming Symposium (ILPS'92)*, Cambridge (MA), 1992. MIT Press.

29. M. Nagl. Set-theoretic approaches to graph-grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 41–54. Springer Verlag, 1987.

30. J. Peckam and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3):153–189, 1988.

31. J. C. Raoult. On graph rewriting. *Theoretical Computer Science*, 32:1–24, 1984.

32. H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*. Oxford University Press, Oxford, 1987.

33. M. Schrefl, A. M. Tjoa, and R. R. Wagner. Comparison criteria for semantic data models. In C. V. Ramamoorthy, editor, *Proc. Int. Conf. on Data Engineering*, pages 120–125. IEEE, Silver Spring (MD), 1984.

34. A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-oriented specification of databases: An algebraic approach. In P. M. Stocker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB '87)*, pages 107–116, Palo Alto, 1987. Morgan-Kaufmann.

35. G. v. Bochmann, M. Barbeau, M. Erradi, P. Lecomte, P. Mondain-Monval, and N. Williams. Mondel: An object-oriented specification language. Technical Report Publication 748, Départment d'Informatique et de Recherche Opérationelle, Université de Montréal, 1990.

36. R. Wieringa. Equational specification of dynamic objects. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG 2.6 Working Conference, Windermere (UK) 1990*, pages 415–438. North-Holland, 1991.

37. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 677–788. North-Holland, Amsterdam, 1990.