The Verigraph System for Graph Transformation

Guilherme Grochau Azzi, Jonas Santos Bezerra, Leila Ribeiro, Andrei Costa, Leonardo Marques Rodrigues, and Rodrigo Machado(✉)

Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil
{ggazzi,jsbezerra,leila,acosta,lmrodrigues,rma}@inf.ufrgs.br

Abstract. Graph transformation (GT) is a rule-based framework, suitable for modelling both static and dynamic aspects of complex systems in an intuitive yet formal manner. The algebraic approach to GT is based on category theory, allowing the instantiation of theoretical results to multiple graph-like structures (e.g. labelled or attributed graphs, Petri nets, even transformation rules themselves). There exists a rich theory of algebraic GT which underlies verification techniques such as static analysis. Current tools based on GT are implemented in a very concrete way, unlike the theory, making them hard to extend with novel theoretical results. Thus a new software system called *Verigraph* was created, with the goal of implementing the theory as closely as possible, while maintaining a reasonable execution time. In this paper we present the architecture of Verigraph, which enabled an almost direct implementation of the theory. We also provide a step-by-step guide to implementing a new graph model within the system, using second-order graph transformation as an example. Finally, we compare the performance of Verigraph and AGG.

Keywords: Graph transformation · Software system · Static analysis

1 Introduction

Graph transformation is a rule-based framework, suitable for modelling both static and dynamic aspects of complex systems in an intuitive yet formal manner [8,24]. The main idea is to use graphs to specify the states of a system, describing existing entities and their relations at each time of execution, and to model the transitions between such states as *graph rewriting rules*, also called *productions*. These rules describe precisely how the states can be modified. Besides having an intuitive and visual representation, graph transformation has a solid formal background, which enables several analysis techniques.

There are several approaches to describe Graph Transformation (GT) [24], differing on the kinds of graphs that are used and how rules and their application are defined. In some approaches, these notions are defined using set theory. The algebraic approach to graph transformation [8] uses notions of category theory to describe graph transformation rules and rule application. Category theory

© Springer International Publishing AG, part of Springer Nature 2018
R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 160–178, 2018.
https://doi.org/10.1007/978-3-319-75396-6_9

provides a language to describe and reason about complex situations at a high level of abstraction. An advantage of this approach is that great part of the rich algebraic GT theory is applicable not only to a particular kind of graph, but also to several different structures such as labelled graphs, typed graphs, attributed graphs [8] and even transformation rules themselves [16]. This is possible since most of the theory is developed at the categorial level, as high-level replacement systems [11] or $\mathcal{M}$-adhesive categories [10]. The main idea is that theory is developed at a very abstract level assuming that the concrete category to which the theory should hold has particular properties. Then, by showing that specific graph categories have these properties, the theory immediately applies to them.

Although most of the algebraic GT theory is done at this abstract level, existing GT tools (AGG [26], Groove [22], among others), are implemented at a rather concrete level: each tool supports only a fixed set of concrete graph models, operations and analysis techniques. Furthermore, their implementation is very far from the formal definitions, hindering arguments about correctness and the construction of extensions to deal with novel approaches.

This led to the creation of the Verigraph System [7], which is open source[1] and implemented in Haskell. It has a current focus on static analysis techniques and the following design goals:

G1. Quick prototyping and experimentation of novel theory
G2. Easy integration of different graph models
G3. Direct implementation of formal concepts at a high level of abstraction, making it easier to reason about correctness
G4. Reasonable execution time.

In this paper we detail the architecture of Verigraph, explaining how the separation of applications (e.g. simulation, static analysis techniques) from concrete graph models (e.g. simple directed graphs, typed graphs, attributed graphs) allows us to achieve goals G1–G3. We also provide a step-by-step guide to implementing custom graph models within the framework, using second-order graph transformation as an example, and provide further evaluation of Verigraph's performance as evidence that goal G4 was achieved.

This paper is organized as follows. Section 2 reviews the theory of algebraic graph transformation. Section 3 presents Verigraph's architecture. To illustrate the flexibility of the system, Sect. 4 provides a step-by-step guide to implementing a graph model within Verigraph. Section 5 provides an overview of currently implemented applications and graph models. Section 6 lists related tools. Section 7 compares the performance of static analysis techniques in Verigraph and AGG. Section 8 provides final remarks and discusses features that are currently under development.

2 Algebraic Graph Transformation

In this section we briefly review the basic definitions of algebraic graph transformation, according to the Double-Pushout (DPO) approach [9]. We follow the

[1] Source code available at https://github.com/Verites/verigraph/.

generalization of DPO to work with objects of any $\mathcal{M}$-adhesive category [10], which include variations of graphs (typed, labelled, attributed), Petri nets and algebraic specifications. The reader is assumed to be familiar with basic concepts of Category Theory. A more detailed introduction to algebraic graph transformation is available in [8].

We begin by reviewing the notion of $\mathcal{M}$-adhesive category, which underlies the other definitions.

Definition 1 ($\mathcal{M}$-adhesive Category). *A category* $\mathbf{C}$ *is called* $\mathcal{M}$*-adhesive, where* $\mathcal{M}$ *is a suitable [10] class of monomorphisms, if*

(i) $\mathbf{C}$ *has pushouts along* $\mathcal{M}$*-morphisms;*
(ii) $\mathbf{C}$ *has pullbacks along* $\mathcal{M}$*-morphisms;*
(iii) pushouts along monomorphisms are van Kampen (VK) squares [14].

These properties ensure that $\mathbf{C}$ has well-behaved pushouts. This equips the category with natural notions for union and intersection of $\mathcal{M}$-subobjects, since it ensures for all objects of $\mathbf{C}$ that their $\mathcal{M}$-subobjects form a distributive lattice. Furthermore, it ensures uniqueness of pushout complements, as described below.

Definition 2 (Pushout Complement)

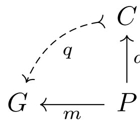
Given the morphisms $(A \xrightarrow{f} B \xrightarrow{g} C)$ *of a category* $\mathbf{C}$*, a pushout complement of* (f, g) *is a pair of morphisms* $(A \xrightarrow{f'} B' \xrightarrow{g'} C)$ *making the square on the right a pushout.*

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g'}\downarrow & & \downarrow{\scriptstyle g} \\ B' & \xrightarrow[f']{} & C \end{array}$$

Fact 1 (Uniqueness of Pushout Complements). *In an* $\mathcal{M}$*-adhesive category* $\mathbf{C}$*, pushout complements along* $\mathcal{M}$*-morphisms (i.e. when* $f \in \mathcal{M}$ *in the square above) are unique up to isomorphism, when they exist.*

We proceed by reviewing the basic concepts of DPO rewriting for any $\mathcal{M}$-adhesive category $\mathbf{C}$.

Definition 3 (Negative Condition)
A **negative condition** *has the form* $\text{NC}(a)$*, where* $a : P \to C$ *is an arbitrary morphism. We say that a morphism* $m : P \to G$ satisfies the condition *when there is no monomorphism* $q : C \rightarrowtail G$ *with* $q \circ a = m$*, i.e. factoring* m *through* a*. We denote by* $\text{NC}(A)$ *a set of negative conditions, where* A *is a set of morphisms.*



Definition 4 (Double-Pushout Rule). *A* **rule***, also called* production*,* $\rho = (L \xleftarrow{l} K \xrightarrow{r} R,\ \text{NC}(N))$ *contains a span in* $\mathbf{C}$ *with* $l, r \in \mathcal{M}$*, as well as a set of negative conditions* $N = \{n_i : L \to N_i\}_{i \in I}$*. We call* L *and* R *the* left- *and* right-hand sides*, respectively, while* K *is called the* interface*. The conditions* $\text{NC}(N)$ *are referred to as* Negative Application Conditions *(NACs).*

Definition 5 (Match and Transformation Step). *A* **match** *for the rule* $\rho = (L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R,\ \textsc{nc}(N))$ *in the object* G *is any morphism* $m : L \to G$. *A match is* applicable *if it satisfies all* NACs *and* (l, m) *has a pushout complement* $(K \overset{k}{\to} D \overset{l'}{\to} G)$.

Given an applicable match $m : L \to G$ *for rule* ρ, *we obtain the* **transformation step** *or* derivation $G \overset{\rho,m}{\Longrightarrow} H$ *by the diagram on the right, where both squares are pushouts.*

$$\begin{array}{ccccc} L & \overset{l}{\longleftarrowtail} & K & \overset{r}{\rightarrowtail} & R \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle k} & & \downarrow{\scriptstyle m'} \\ G & \underset{l'}{\longleftarrowtail} & D & \underset{r'}{\rightarrowtail} & H \end{array}$$

Definition 6 (Double-Pushout Transformation System). *A* DPO ***transformation system*** *(TS) in the* $\mathcal{M}$*-adhesive category* $\mathbf{C}$ *is a tuple* $\mathcal{G} = (G_0, P)$ *where* G_0 *is a* $\mathbf{C}$*-object, representing the initial state, and* P *a set of rewriting rules.*

The categorial foundation for DPO has enabled the definition of multiple analyses that are also applicable to any $\mathcal{M}$-adhesive category. In particular, critical pair analysis helps understand the control flow that emerges from interacting rules [11]. It is based on parallel independence.

Parallel independence captures the notion that two transformation steps do not interfere with each other, being applicable in any order while still reaching the same result. When they are not parallel-independent, at least one of the steps disables the application of the other, which is called a conflict. In the following discussion, we omit the treatment of NACs due to limited space.

Definition 7 (Parallel Independence). *Given two transformation steps* $G \overset{\rho_1,m_1}{\Longrightarrow} H_1$ *and* $G \overset{\rho_2,m_2}{\Longrightarrow} H_2$, *they are* **parallel-independent** *if there exist morphisms* $q_{12} : L_1 \to D_2$ *and* $q_{21} : L_2 \to D_1$ *making the following diagram commute. The rules are said to be in* **conflict** *when they are not parallel-independent.*

$$\begin{array}{ccccccccc} R_1 & \overset{r_1}{\longleftarrow} & K_1 & \overset{l_1}{\longrightarrow} & L_1 & \cdots\cdots & L_2 & \overset{l_2}{\longleftarrow} & K_2 & \overset{r_2}{\longrightarrow} & R_2 \\ \downarrow & & \downarrow & {\scriptstyle q_{21}} & {\scriptstyle m_1}\searrow & & \swarrow{\scriptstyle m_2} & {\scriptstyle q_{12}} & \downarrow & & \downarrow \\ H_1 & \longleftarrow & D_1 & & \underset{l'_1}{\longrightarrow} & G & \underset{l'_2}{\longleftarrow} & & D_2 & \longrightarrow & H_2 \end{array}$$

Remark 1. The conflicts characterized by the definition above are called **delete-use** conflicts, because in the context of graphs they detect elements that are deleted by one rule and used by the other. The complete notion of parallel independence in the presence of NACs is more involved, including other kinds of conflicts. A thorough treatment of these notions may be found in [15].

The notion of critical pair captures conflicts in a minimal context. By enumerating all critical pairs, we get an account of all possible interference between two rules. Their formal definition and a thorough discussion is provided in [11].

The categories of graphs and of typed graphs, which are now introduced, are $\mathcal{M}$-adhesive. Thus, the generalized theory of DPO transformation applies to those categories.

Definition 8 (Graph, Graph Morphism). *A* **graph** $G = (V, E, s, t)$ *contains a set* V *of nodes, a set* E *of edges and two functions* $s, t : E \to V$ *mapping each edge into its source and target node, respectively. Given graphs* $G_1 = (V_1, E_1, s_1, t_1)$ *and* $G_2 = (V_2, E_2, s_2, t_2)$*, a* **graph morphism** $f : G_1 \to G_2$ *is a pair of functions* $f = (f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ *that preserve incidence, that is,* $f_V \circ s_1 = s_2 \circ f_E$ *and* $f_V \circ t_1 = t_2 \circ f_E$*.*

Definition 9 (T-typed Graph, T-typed Graph Morphism). *Given a graph* T*, called the* type graph*, a* T-**typed graph** *is a pair* (G, t) *where* G *is the* instance graph *and* $t : G \to T$ *the* typing morphism*. Given two* T*-typed graphs* (G_1, t_1) *and* (G_2, t_2)*, any graph morphism* $f : G_1 \to G_2$ *that preserves typing, i.e. with* $t_2 \circ f = t_1$ *is a* T-**typed graph morphism**.

Definition 10 (Categories of Graphs). *Graphs along with graph morphisms form the category* **Graph**. T*-typed graphs along with their morphisms form the category* $\mathbf{Graph}_T$*. Note that* $\mathbf{Graph}_T$ *is the slice category* $\mathbf{Graph} \downarrow T$*.*

Fact 2. *The categories* **Graph** *and* $\mathbf{Graph}_T$ *are* $\mathcal{M}$*-adhesive, taking as* $\mathcal{M}$ *the class of all monomorphisms [10].*

2.1 Example: Pacman

In this paper we will use the Pacman game as a running example, adapted from [24], especially to discuss second-order graph transformation in Sect. 4. The example is depicted in Fig. 1. We use a typed graph transformation system having 4 types of nodes and 5 types of edges (graph T). Rules describe how Pacman and the ghosts may move (rules movePacman and moveGhost); how a ghost may kill Pacman (rule killPacman, that has a NAC – graph with gray background – stating that Pacman may only be killed if it does not carry a berry); how Pacman may kill a ghost (rule killGhost); and how Pacman may get and drop berries (rules getBerry and dropBerry). Note that only the left- and right-hand sides of rules are shown since in these examples all items that are shown in both sides are preserved (thus graph K is their intersection and rule morphisms are obvious).

3 Architecture

Verigraph was implemented in Haskell [19], a purely functional programming language with mathematical foundations. We exploit its abstraction mechanisms and functional style to reduce the mismatch between theory and code as much as possible.

The core of the system is organized into three layers: The *Abstract* layer is the central part of the architecture, providing high-level categorial and rewriting APIs. The *Application* layer provides a series of analysis techniques mainly implemented over the abstract APIs. Finally, the *Concrete* layer encapsulates the realisation of those APIs for specific categories, such as **Graph** and $\mathbf{Graph}_T$.

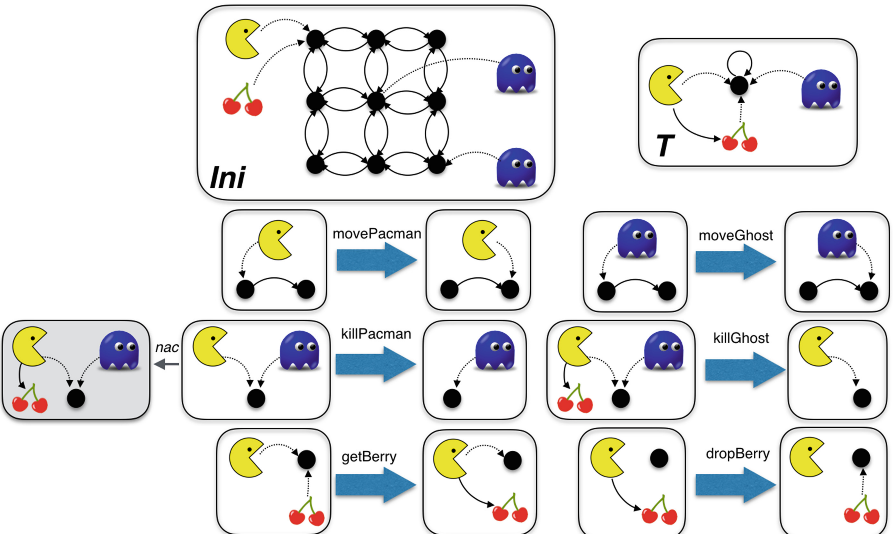


Fig. 1. Pacman transformation system

Figure 2 summarizes the architecture of Verigraph. Folders represent Haskell modules, while simple boxes represent data types or, when text is in italic, type classes. Dashed arrows indicate dependencies between modules. A thorough documentation of Verigraph's modules is available online[2].

In the remainder of this section, we explain the design of each layer in detail, and how they are used to achieve a generic and extensible architecture that is also very close to theory.

3.1 The Abstract Layer

The abstract layer is responsible for defining basic constructions and operations for category theory and rewriting systems. This is mainly accomplished by a series of contracts, in the form of Haskell type classes, that specify abstract categorial operations. The application layer can then be largely generic with respect to the category, depending only on these contracts. The actual implementation of most operations is left for the concrete layer, though some operations have a default implementation in terms of other categorial constructs (e.g. pushout as coproduct and coequalizer).

Being the layer that directly expresses categorial concepts, its operations closely reflect categorial definitions. In Verigraph, a category is described by the type class `Category`, shown in Fig. 3. It defines the basic structure and operations that any category implemented in Verigraph must provide. We omit some details of the type classes due to space restrictions.

[2] API documentation available at https://verites.github.io/verigraph-docs/.

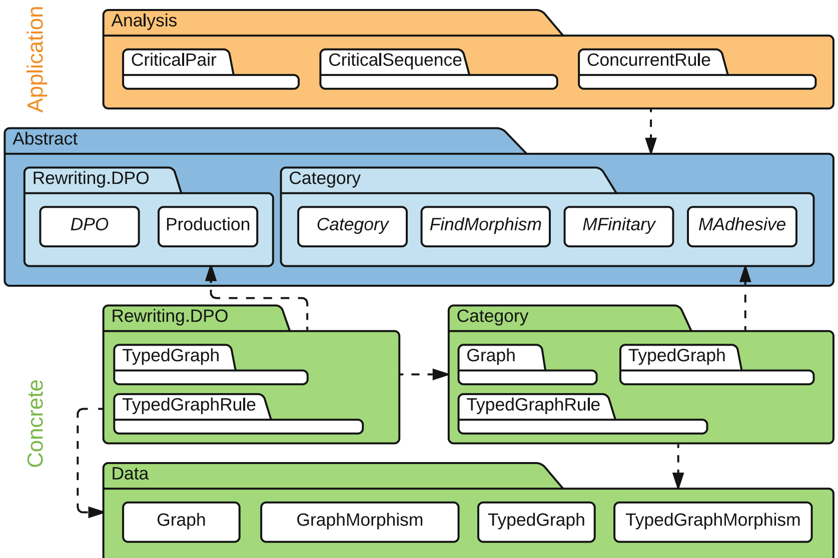


Fig. 2. Verigraph architecture

Essentially, a category must have an *object type*[3], a *morphism composition* operation (`<&>`), a function that returns the *identity* morphism of an object, together with functions that return the *domain* and *codomain* objects of a morphism.

However, not all categories are suitable for all purposes. In general, the theory of rewriting assumes that categories have enough structure to enable particular constructions. This is also specified by type classes, such one for $\mathcal{M}$-adhesive categories, which have pushouts and pullbacks along monomorphisms, as shown in Fig. 3.

The categorial portion of this module provides further type classes. For example, `FindMorphism` defines the operations for finding all morphisms between a pair of objects, possibly restricting to a specific class or satisfying restrictions (e.g. make a span or a cospan commute). Similarly, `E'PairCofinitary` deals with enumerating jointly epic pairs of morphisms with given domains.

Besides the high-level categorial framework, the abstract layer also contains APIs for each rewriting approach. This is kept in a different submodule to decouple categorial operations from the rewriting approaches.

Currently, the only rewriting approach implemented in the stable version of Verigraph is DPO, although new approaches are being studied and have ongoing implementation, particularly Sesqui-Pushout [6] and AGREE [5].

[3] We use an extension of Haskell's type system enabling *type families* to associate a type of object to each type of morphism.

```
class Eq morph => Category morph where
    type Obj morph :: *
    (<&>)    :: morph -> morph -> morph
    identity :: Obj morph -> morph
    domain   :: morph -> Obj morph
    codomain :: morph -> Obj morph

class Category morph => MAdhesive morph where
  calculatePushoutAlongM            :: morph -> morph -> (morph, morph)
  calculatePullbackAlongM           :: morph -> morph -> (morph, morph)
  hasPushoutComplementAlongM        :: morph -> morph -> Bool
  calculatePushoutComplementAlongM :: morph -> morph -> (morph, morph)
```

Fig. 3. Type classes for categories and $\mathcal{M}$-adhesive categories.

```
data Production morph = Production
  { left  :: morph
  , right :: morph
  , nacs  :: [morph] }
```

Fig. 4. Data type for DPO productions (see Definition 4)

```
calculateDPO :: MAdhesive morph => morph -> Production morph
                -> (morph, morph, morph, morph)
calculateDPO m (Production l r _) =
  let (k, l') = calculatePushoutComplementAlongM l m
      (r', m') = calculatePushoutAlongM r k
  in (k, m', l', r')
```

Fig. 5. Implementation of DPO rewriting steps (see Definition 5)

Similar to the categorial APIs, those for rewriting provide basic data structures and operations of their underlying approaches, often defined in terms of categorial operations. Examples are data types for rules and transformation systems, along with functions for checking if matches are applicable, performing rewritings, etc. Figure 4 shows the data definition for productions in the DPO approach and Fig. 5 shows a function that, given a production and an applicable match, calculates the transformation step.

3.2 The Application Layer

The application layer provides a collection of analysis techniques, mainly based on the categorial and rewriting APIs. An example is *critical pair analysis*,

described in Sect. 2. It is implemented at a high level of abstraction, directly depending only on the DPO and E'PairCofinitary type classes (which in turn depend on FindMorphism).

Thus, critical pair analysis is available for any category that conforms to the aforementioned type classes. Figure 6 shows the main functions that implement critical pair analysis. Once again, we omit the treatment of NACs due to limited space. The code also has been slightly simplified for readability and space.

```
isDeleteUse :: (DPO morph, E'PairCofinitary morph) =>
    Production morph -> (morph, morph) -> Bool
isDeleteUse p1 (m1,m2) =
  let (_,l1') = calculatePushoutComplementAlong (leftMorphism p1) m1
  in null (findCospanCommuters m2 l1')

findAllDeleteUse :: (DPO morph, E'PairCofinitary morph) =>
    Production morph -> Production morph -> [(morph, morph)]
findAllDeleteUse p1 p2 =
  let
    allPairs = createJointSurjections (leftMorphism p1) (leftMorphism p2)
    satisfyingPairs = filter (satisfyRewritingConditions p1 p2) allPairs
  in filter (isDeleteUse p1) satisfyingPairs
```

$$R_1 \xleftarrow{r_1} K_1 \xrightarrow{l_1} L_1 \qquad L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2$$

$$K_1 \rightarrow D_1, \quad q_{21} : L_2 \dashrightarrow D_1, \quad m_1 : L_1 \rightarrow G, \quad m_2 : L_2 \rightarrow G, \quad D_1 \xrightarrow{l_1'} G$$

Fig. 6. Functions that calculate critical pairs for delete-use (see Definition 7 and Remark 1)

In the code snippet shown in Fig. 6, the function findJointSurjections enumerates all jointly epic pairs of morphisms that have the left-hand sides of either rule as domain. Function satisfyRewritingConditions tests the existence of pushout complements for a pair of matches. Function findCospanCommuter finds all morphisms $q_{21} : L_2 \rightarrow D_1$ such that $g_1 \circ q_{21} = m_2$.

Although the main focus of this layer is to be generic and based on the abstract layer, there exist analyses that are tied to the internal structure of a particular category and/or its objects. For such situations, Verigraph allows the application layer to directly access features of the concrete layer, at the cost that these particular analyses will not be available to other categories. Some examples are given in Sects. 4 and 5.

3.3 The Concrete Layer

The concrete layer deals with the implementation of particular categories, i.e. the data structures and category-specific operations for objects and morphisms (e.g.

applying a morphism to a node or edge), as well as the realisation of contracts from the abstract layer, such as operations for finding and composing morphisms, pushouts and pullbacks.

This layer is split into three modules, as shown in Fig. 2. The `Data` module provides basic data structures, while the `Category` and `Rewriting` modules provide instantiations of the appropriate type classes for those data structures.

Currently, there are three main categories implemented in Verigraph: **Graph**, $\mathbf{Graph}_T$ (see Definition 10) and $\mathbf{GraphRule}_T$ (which will be described in Sect. 4). A category for typed attributed graphs is under development. Figure 7 shows data structures representing objects and morphisms of the first two categories.

In order to use the existing generic analyses with a different category, this is the only layer that needs to be changed, as explained in Sect. 4. One of the main advantages is that new programmers do not have to worry about too many categorial details, as long as they implement the basic operations defined in the upper layers. Figure 8 shows the instantiation of category $\mathbf{Graph}_T$.

Another advantage of this architecture is that optimizations dependent on the internal representation of objects and morphisms can be done here without

```
data Graph n e = Graph
    { nodeMap :: [(NodeId, Node n)]
    , edgeMap :: [(EdgeId, Edge e)] }

data GraphMorphism a b = GraphMorphism
  { domainGraph   :: Graph a b
  , codomainGraph :: Graph a b
  , nodeRelation  :: Relation NodeId
  , edgeRelation  :: Relation EdgeId }

type TypedGraph a b = GraphMorphism a b

data TypedGraphMorphism a b = TypedGraphMorphism
  { domainGraph   :: TypedGraph a b
  , codomainGraph :: TypedGraph a b
  , mapping       :: GraphMorphism (Maybe a) (Maybe b) }
```

Fig. 7. Implementation of typed graphs and their morphisms

```
instance Category (TypedGraphMorphism a b) where
  type Obj (TypedGraphMorphism a b) = TypedGraph a b
  domain     = domainGraph
  codomain   = codomainGraph
  t2 <&> t1  = TypedGraphMorphism (domainGraph t1) (codomainGraph t2)
                                  (mapping t2 <&> mapping t1)
  identity t = TypedGraphMorphism t t (identity (domain t))
```

Fig. 8. `Category` instance for $\mathbf{Graph}_T$

compromising the clarity of the abstract operations or tying them to a particular category. An example are optimized search procedures for morphisms that satisfy particular restrictions, such as commuting with a particular span or cospan.

Finally, despite being focused on graphs, Verigraph is not necessarily limited to them. Therefore, any category with the necessary properties can be implemented in the system, such as sets, Petri nets, algebraic specifications, etc.

4 Implementing a Graph Model in Verigraph

In this section, we provide a step-by-step guide to implementing a new graph model in Verigraph. Note that all implementations shown in this section belong to the concrete layer. Each step will be illustrated by describing the implementation of Second-Order Graph Grammars (SOGGs), which allow the transformation of graph rewriting rules using DPO transformation [17]. In the context of Model-Driven Engineering, SOGGs are well-suited to analyse changes introduced during the evolution or maintenance phase of development.

4.1 Step 1: Define the Graph Model as an $\mathcal{M}$-adhesive Category

In order to integrate a graph model into Verigraph's architecture, it must be defined as an $\mathcal{M}$-adhesive category. The notions of object and morphism must be clear, as well as constructions such as (co)limits, initial pushouts, $\mathcal{E}$-$\mathcal{M}$ factorization and $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

Note that most of the implemented categories are finitary, that is, each object has a finite number of subobjects. In this case, $\mathcal{M}$-adhesiveness guarantees $\mathcal{E}$-$\mathcal{M}$ factorization and the existence of initial pushouts, and a strict initial object additionally guarantees finite coproducts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization [4].

Second-order graph transformations are defined in the category of typed graph spans, which is $\mathcal{M}$-adhesive under certain assumptions [16].

Graph rewriting rules, in the DPO approach, are spans $L \leftarrowtail K \rightarrowtail R$ in $\mathbf{Graph}_T$. Thus, the following category is appropriate to model the rewriting of graph rules.

Definition 11. *The category* $\mathbf{GraphRule}_T$ *has, as objects, monic spans* $L \leftarrowtail K \rightarrowtail R$ *of* $\mathbf{Graph}_T$*. A* **rule morphism** $f : \alpha \to \beta$ *is then a triple of graph morphisms* (f_L, f_K, f_R) *between the graphs of both rules making the diagram on the right commute.*

$$\begin{array}{ccccc} L_\alpha & \xleftarrow{l_\alpha} & K_\alpha & \xrightarrow{r_\alpha} & R_\alpha \\ \downarrow f_L & & \downarrow f_K & & \downarrow f_R \\ L_\beta & \xleftarrow[l_\beta]{} & K_\beta & \xrightarrow[r_\beta]{} & R_\beta \end{array}$$

Fact 3. $\mathbf{GraphRule}_T$ *is* $\mathcal{M}$*-adhesive [16]. Its limits, colimits and pushout complements can be constructed componentwise in* $\mathbf{Graph}_T$*.*

Since $\mathbf{GraphRule}_T$ is $\mathcal{M}$-adhesive, the framework of DPO rewriting can be instantiated for it. Thus, ***second-order rules***, also called *2-rules*, are spans of

rule morphisms. Figure 9 shows two examples of 2-rules for the Pacman transformation system. Note that the interfaces are omitted from first- and second-order rules, since all items shown in both sides are preserved (thus the interfaces are the intersection of the left- and right-hand sides). The 2-rule *noViolence* transforms a deletion of *Pacman* into its preservation, maintaining the deletion of the edge linking *Pacman* to a *block*. The 2-rule *fastGhost* adds a new preserved *block* in a rule that moves a *ghost*, allowing it to move two blocks at a time instead of just one.

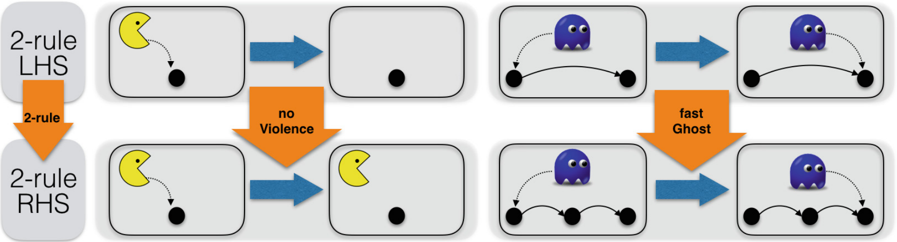


Fig. 9. Pacman second-order rules

4.2 Step 2: Implement Data Structures for Objects and Morphisms

The building blocks for DPO transformation are the objects and morphisms of the category. Thus, the design of data structures that represent them is crucial to ensure reasonable runtime and memory consumption.

The category **GraphRule**$_T$ is particularly simple, since it can reuse the data structures that implement **Graph**$_T$. The types of graph rules and their morphisms are show in Fig. 10, and they directly reflect Definition 11.

```
type TypedGraphRule n e = Production (TypedGraphMorphism n e)

data RuleMorphism n e = RuleMorphism
  { rmDomain          :: TypedGraphRule n e
  , rmCodomain        :: TypedGraphRule n e
  , mappingLeft       :: TypedGraphMorphism n e
  , mappingInterface :: TypedGraphMorphism n e
  , mappingRight      :: TypedGraphMorphism n e }
```

Fig. 10. `RuleMorphism` implementation

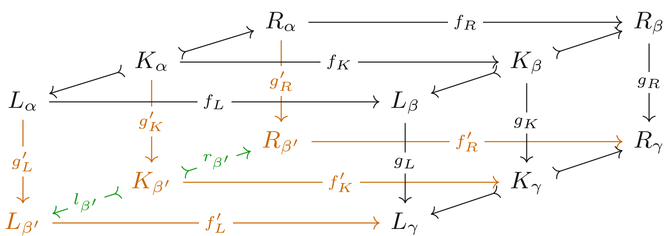
4.3 Step 3: Instantiate the Appropriate Type Classes

According to the constructions available in the category, the type classes of the abstract layer should be instantiated, that is, their operations should be

implemented for the objects and morphisms of this category. The instantiated type classes will define which applications are available to the implemented graph model. The correspondence between each type class and the categorial concepts it embodies are explained in the API documentation[4].

Many of the type classes provide default implementations for some of the operations, which are correct but inefficient. These are useful for early prototypes, but careful implementation of all operations is important to ensure reasonable runtime and memory consumption.

In the case of $\mathbf{GraphRule}_T$ all of the aforementioned type classes are instantiated. Indeed, most of the algorithms are constructed over $\mathbf{Graph}_T$ primitives since limits, colimits and other constructions on $\mathbf{GraphRule}_T$ can be calculated componentwise. An example is the implementation of pushout complements, shown in Fig. 11. The function receives two `RuleMorphisms` $f : \alpha \to \beta$




```
instance MAdhesive (RuleMorphism a b) where
  calculatePushoutComplementAlongM
    g@(RuleMorphism _ ruleC fL fK fR)
    f@(RuleMorphism ruleA ruleB gL gK gR) =
    let
      (gL', fL') = calculatePushoutComplementAlongM fL gL
      (gK', fK') = calculatePushoutComplementAlongM fK gK
      (gR', fR') = calculatePushoutComplementAlongM fR gR

      leftB'  = commutingMorphismSameCodomain
                (leftMorphism ruleC <&> fK') fL'
                 gK' (gL' <&> leftMorphism ruleA)
      rightB' = commutingMorphismSameCodomain
                (rightMorphism ruleC <&> fK') fR'
                 gK' (gR' <&> rightMorphism ruleA)

      ruleB' = Production leftB' rightB' []
    in
      ( RuleMorphism ruleA ruleB' gL' gK' gR',
        RuleMorphism ruleB' ruleC fL' fK' fR' )
```

Fig. 11. Implementation of *Pushout Complement* for `RuleMorphism`

[4] API documentation available at https://verites.github.io/verigraph-docs/.

and $g : \beta \to \gamma$, returning its pushout complement $(f' : \alpha \to \beta',\ g' : \beta' \to \gamma)$ (see Definition 2). The first part of the function performs pushout complements in $\mathbf{Graph}_T$, shown in orange in the previous diagram. This implicitly constructs the typed graphs of rule β' as codomains of g'_L, g'_K and g'_R. The second part searches for graph morphisms $l_{\beta'}$ and $r_{\beta'}$ that make the diagram commute. Finally, the rule β' and the rule morphisms f' and g' are assembled from their components.

The complete realisation of `MAdhesive`, `DPO` and `E'PairCofinitary` type classes for `RuleMorphism` enables the usage of many generic applications of Verigraph with 2-rules, including critical pair analysis and the calculation of concurrent rules.

4.4 Step 4 (Optional): Implement Category-Specific Applications

There are often applications that depend on details of the graph model, and cannot or were not generalized categorially. These may be still be implemented within Verigraph, although they may not profit from the separation of the abstract layer.

In the case of SOGGs, inter-level critical pair analysis detects situations where the applicability of first-order rules is changed after applying a second-order rule. This operation is inherent to the transformation of graph rules, and was implemented in Verigraph.

4.5 Step 5: Adapt the Command-Line Interface

In order for end users to execute applications over the new graph model, some functionality must still be implemented, such as: reading files that define transformation systems of the graph model, writing files that describe the results, interpreting configuration options. This needs to be implemented separately for each graph model, since the syntax of input/output files and the available options vary greatly. It also needs to be integrated into the executable providing the command-line interface. This step is unrelated to the theory of algebraic graph transformation, so it is beyond the scope of this paper.

5 Overview of Implemented Techniques

Verigraph already provides several applications for end users. It allows the execution of first- and second-order rewriting rules over typed graphs (i.e. transformation of typed graphs and of graph rules). Its current main focus, however, are static analysis techniques. Attributed graphs are not yet supported. At the present time, Verigraph provides a Command Line Interface, using AGG [26] `.ggx` and `.cpx` files as input/output formats.

Many of the implemented features are not specific to first- and second-order graph transformation, being implemented generically with respect to the transformed structure. This includes the execution of second-order rules, which may be applied to first-order transformation of any structure. Other generic applications are the following static analysis techniques described in [8].

Critical Pair Analysis: Captures all possibilites of conflicts between rules in a minimal context.

Critical Sequence Analysis: Similar to critical pair analysis, captures all possibilites of dependencies between rules in a minimal context.

Concurrent Rules Calculation: Generates rules that summarize the application of several rules in a single step.

As for the applications available to a single kind of structure, Verigraph provides:

Inter-level Critical Pairs Analysis (GraphRule$_T$) [17]: Detects conflicts between first- and second-order graph rules, i.e. whether and how the application of second-order rules affects the applicability of first-order rules. This is useful for analysing software evolution.

Occurrence Grammar Calculation (Graph$_T$) [23]: Generates doubly-typed graph grammars that describe the semantics of typed graph grammars and their application history. These can be used for the generation of test cases.

Another important aspect of DPO transformation systems are Graph Constraints, which are also supported by Verigraph. They are not described in this paper due to lack of space.

6 Related Work

Existing tools based on graph transformation (GT) vary according to the supported graph models, transformation approaches, execution models and analysis techniques. Unlike Verigraph, most of them are domain-specific, but other domain-neutral tools based on the algebraic approach include the following.

AGG [26] supports typed attributed graphs, allowing execution of transformation rules, critical pair analysis and calculation of concurrent rules.

GROOVE [22] supports a model of labelled graphs with types and attributes, allowing execution of transformation rules, state space exploration and model checking.

GrGEN.NET [12] supports typed attributed graphs, allowing the compilation of transformation rules into C# for efficient execution. It also provides a domain-specific language for controlling the application of rewrite rules.

Graph Programs [18] is a programming language containing transformation rules of labeled graphs as primitive statements. A compiler is available, generating bytecode for the York abstract machine. Although a reasoning system based on Hoare logic was proposed for verifying Graph Programs [21], tool support is not yet available.

A particular application domain where many transformation tools are available is Model-Driving Engineering (MDE). In this setting, transformation rules generally manipulate graph-based models such as those from the Unified Modeling Language (UML) or the Eclipse Modeling Framework (EMF). Besides DPO

rewriting, Triple Graph Grammars (TGG) [25] are often used to define bidirectional model transformations. **EMorF** [13] supports in-place model modification rules (based on DPO), as well as model transformation and synchronization (based on TGG) for EMF models, allowing execution of transformation rules. **eMoflon** [1] supports Story Driven Modeling (a combination of UML Activity Diagrams and DPO) as well as TGG for EMF models, allowing compilation into Java code. **Henshin** [2] supports transformation rules for EMF models, along with control-flow constructs to guide their execution. It allows execution of rules, compilation into Java, state space exploration integrated with model-checking tools and critical pair analysis (using AGG as a component).

7 Performance Evaluation

One of the design goals of Verigraph was a reasonable execution time. In order to validate the satisfaction of this goal, we have performed some experiments comparing the execution time of static analysis techniques on Verigraph and AGG [26].

We have selected five transformation systems (TS) to use as inputs of the experiment. PACMAN is the TS presented in Fig. 1. ELEV models the behaviour of an elevator system with 9 rules, adapted from [15]. MED1, MED2 and MED3 model guidelines for a medical procedure, containing 36 rules in total [3]. PACMAN and MED1 contain only relatively small graphs, with at most 5 nodes, and at most 2 of the same type. ELEV contains slightly larger graphs, with up to 7 nodes and 3 of the same type. MED2 and MED3 have even larger graphs, with up to 8 nodes and 2 of the same type.

The experiment consisted of running critical pair and sequence analysis over the TSs, that is, calculating all critical pairs and sequences for each pair of rules in each TS. The analysis was executed 10 times for each TS, with each tool, in an Intel I5-3330 processor running at 3 GHz with 16 GiB of RAM. The total time and peak memory usage was measured for each execution using GNU `time`. Although the measurements include reading and writing XML files, this overhead should be similar for both tools, and both execution time and memory usage should be dominated by the analysis itself. Furthermore, the results directly reflect the experience of end users. It is also important to note that both AGG and Verigraph parallelize the analysis. All input files and the scripts used for running the tests are available at https://github.com/Verites/verigraph/tree/critical-pairs-benchmarks.

Table 1 presents the number of critical pairs and sequences found for each TS, and Table 2 presents the measured execution times. For the TSs PACMAN, MED1 and ELEV, Verigraph's performance was slightly better than AGG. On MED2 and MED3 AGG's performance degraded substantially, and it was significantly outperformed by Verigraph. This indicates that AGG is more sensitive to the size of the graphs contained in rewriting rules, since the last two TSs contained the largest graphs. Memory usage is shown in Table 3, and followed a similar pattern.

Table 1. Number of critical pairs and sequences for each transformation system.

	PACMAN	ELEV	MED1	MED2	MED3
Rules	6	9	9	11	12
Critical pairs	135	328	85	858	1462
Critical sequences	146	242	16	229	272

Table 2. Average and standard deviation for execution times of critical pair and sequence analysis, in seconds.

Tool	PACMAN		ELEV		MED1		MED2		MED3	
	avg	dev	avg	dev	avg	dev	avg	dev	avg	dev
Verigraph	3.82	0.07	13.21	0.09	3.55	0.06	14.94	0.18	35.84	0.30
AGG	4.90	0.05	15.16	0.12	5.49	0.05	1233.10	12.91	1375.62	16.43

Table 3. Average and standard deviation for peak memory usage of critical pair and sequence analysis, in MiB.

Tool	PACMAN		ELEV		MED1		MED2		MED3	
	avg	dev	avg	dev	avg	dev	avg	dev	avg	dev
Verigraph	65.50	2.27	157.88	4.12	47.37	2.02	434.34	15.70	764.60	23.52
AGG	180.70	7.66	287.00	4.74	112.65	1.10	8660.15	299.06	8739.41	229.26

8 Conclusion

Verigraph is a new system for Graph Transformation (GT) that exploits the use of category theory to promote flexibility and extensibility. The use of category theory as a basis allows not only for flexibility, but also provides a framework in which formal definitions of algebraic GT can be implemented in a rather straightforward way. Conceptually, this idea is similar to the use of institutions as a basis for the HETS tool [20].

The flexibility of Verigraph was demonstrated by the implementation of second-order graph transformation: by instantiating a categorial API for $\mathbf{GraphRule}_T$, existing applications like critical pair analysis are automatically available for second-order graph rewriting. Moreover, we have shown that it is also possible to develop category-specific applications, like Interlevel Conflict Analysis (which requires at least two levels of rewriting rules).

Despite flexibility and extensibility, a reasonable execution time can be achieved by an efficient implementation of the categorial API. This was demonstrated by comparing execution time of critical pair analysis on Verigraph and AGG, having Verigraph outperform AGG in realistic test cases.

Currently Verigraph implements rewriting under the DPO approach using the categories $\mathbf{Graph}_T$ and $\mathbf{GraphRule}_T$, as well as critical pair/sequence

analysis and calculation of concurrent rules. We are working on support for (typed) attributed graphs, as well as the Sesqui-Pushout and AGREE transformation approaches. Research is also being done on applying Verigraph to generate test cases for software that is modeled with graph transformation.

The existing version of Verigraph uses AGG files for input and output via a Command Line Interface. Nonetheless, Verigraph's own Graphical User Interface is under development, following a web-based approach completely decoupled from the system's code.

A flexible architecture makes Verigraph suitable as a platform for testing new ideas in Graph Transformation. The system is free and open source, currently available online at GitHub[5], which allows collaborative development and discussion. Extensive automated testing helps maintain its correctness, and its API is thoroughly documented. All these aspects could enable the use and development of Verigraph by the community of researchers.

References

1. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon: leveraging EMF and professional CASE tools. Informatik **192**, 281 (2011)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for In-Place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
3. Bezerra, J.S., Costa, A., Ribeiro, L., Cota, É.F.: Formal verification of health assessment tools: a case study. Electron. Notes Theor. Comput. Sci. **324**, 31–50 (2016)
4. Braatz, B., Ehrig, H., Gabriel, K., Golas, U.: Finitary M-adhesive categories. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 234–249. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2_16
5. Corradini, A., Duval, D., Echahed, R., Prost, F., Ribeiro, L.: AGREE – algebraic graph rewriting with controlled embedding. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 35–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_3
6. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_4
7. Costa, A., Bezerra, J., Azzi, G., Rodrigues, L., Becker, T.R., Herdt, R.G., Machado, R.: Verigraph: a system for specification and analysis of graph grammars. In: Ribeiro, L., Lecomte, T. (eds.) SBMF 2016. LNCS, vol. 10090, pp. 78–94. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49815-7_5
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2

[5] https://github.com/Verites/verigraph/.

9. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: an algebraic approach. In: Switching and Automata Theory, pp. 167–180 (1973)
10. Ehrig, H., Golas, U., Hermann, F., et al.: Categorical frameworks for graph transformation and HLR systems based on the DPO approach. Bull. EATCS **102**, 111–121 (2010)
11. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement categories and systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 144–160. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_12
12. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: a fast SPO-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_27
13. Klassen, L., Wagner, R.: EMorF - a tool for model transformations. Electron. Commun. EASST **54**, 1–6 (2012)
14. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO - Theor. Inf. Appl. **39**(3), 511–545 (2005)
15. Lambers, L.: Certifying rule-based models using graph transformation. Ph.D. thesis, Elektrotechnik und Informatik der Technischen Universität Berlin (2010)
16. Machado, R.: Higher-order graph rewriting systems. Ph.D. thesis, Instituto de Informática - Universidade Federal do Rio Grande do Sul (2012)
17. Machado, R., Ribeiro, L., Heckel, R.: Rule-based transformation of graph rewriting rules: towards higher-order graph grammars. Theor. Comput. Sci. **594**, 1–23 (2015)
18. Manning, G., Plump, D.: The GP programming system. Electron. Commun. EASST **10**, 1–13 (2008)
19. Marlow, S.: Haskell 2010 language report (2010). https://www.haskell.org/onlinereport/haskell2010/
20. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, HETS. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_40
21. Poskitt, C.M.: Verification of graph programs. Ph.D. thesis, University of York (2013)
22. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_40
23. Ribeiro, L.: Parallel composition and unfolding semantics of graph grammars. Ph.D. thesis, Technical University of Berlin (1996)
24. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1, Foundations. World Scientific Publishing Co., Inc., River Edge (1997)
25. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59071-4_45
26. Taentzer, G.: AGG: a tool environment for algebraic graph transformation. In: Nagl, M., Schürr, A., Münch, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 481–488. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45104-8_41