Preface

Graphs of all kinds are widely used to model complex states, structured objects, networks, relational structures, and diagrams in many areas of computer science. Even wider, rules are used to define permitted actions and transitions. Graph transformation combines graphs and rules into a computing paradigm, the theory of which has been well-developed in the last three decades. At the same time, a wide spectrum of potential applications has been studied showing that applied graph transformation provides a rule-based framework for the specification and development of systems, languages, and tools that are founded on graphs. The main topics of interest include the following:

Language issues. Features in graph transformation based languages, like typing, modularity, refinement, parallelism, concurrency, distribution, optimization, and correctness.

Tool issues. Conception and design of support tools for graph transformation based languages including editors, parsers, interpreters, compilers, optimizers, verifiers, and graphical user interfaces.

Application domains. Demonstration of the usefulness of graph transformation by case studies in various areas like definition of visual languages, database models, concurrent and distributed systems, software process modeling, implementation of programming languages.

Readers, who want to know more about the state of the art of graph transformation, may consult the three volumes of the *Handbook of Graph Grammars and Computing by Graph Transformation* published by World Scientific.

The APPLIGRAPH *Workshop on Applied Graph Transformation* (AGT 2002), a satellite event of ETAPS 2002, is intended to be a forum to discuss recent developments in the area. The workshop programme consists of 17 accepted contributions that are documented in these proceedings. The contributions cover a wide range of topics in theory and applications. They are organized into six chapters corresponding to the sessions of the workshop.

AGT 2002 is the final event of the ESPRIT Working Group APPLIGRAPH *(Applications of Graph Transformation)* that started five years ago. The aims of APPLIGRAPH are

- to coordinate and promote research activities in applied graph transformation,
- to improve the systematic exchange and dissemination of information on graph transformation and its applications,
- to provide formal graph transformation methods and modelling tools supporting the software development process including visualisation, prototyping, and safety issues, and
- to provide formal graph transformation methods and modelling tools supporting the specification and implementation of concurrent and distributed systems.

APPLIGRAPH consists of nine teams with Bremen as coordinating site.

- Rheinisch-Westfälische Technische Hochschule Aachen, D (Manfred Nagl)
- Universitaire Instelling Antwerpen, B (Dirk Janssens)
- Technische Universität Berlin, D (Hartmut Ehrig)
- Universität Bremen, D (Hans-Jörg Kreowski)
- Universiteit Leiden, NL (Grzegorz Rozenberg)
- Katholieke Universiteit Nijmegen, NL (Rinus Plasmeijer)
- Universität Paderborn, D (Gregor Engels)
- Università di Pisa, I (Ugo Montanari)
- Università degli Studi di Roma, I (Francesco Parisi-Presicce)

The team leaders and Detlef Plump as APPLIGRAPH manager form the Programme Committee of AGT 2002.

In the name of the Programme Committee, we would like to thank the reviewers (who are not members of the PC) Paolo Baldan, Rosi Bardohl, Andrea Corradini, Ingrid Fischer, Annegret Habel, Berthold Hoffmann, Renate Klempien-Hinrichs, Peter Knirsch, Sabine Kuske, Andrea Maggiolo Schettini, Manfred Muench, Anilda Qemali, and Bernhard Westfechtel for their valuable help. We are also quite grateful to the ETAPS organizers, in particular, to Susanne Graf and Rachid Echahed for hosting AGT 2002 as a satellite workshop. Finally, we would like to acknowledge the financial support by the European Commission.

March 2002 Hans-Jörg Kreowski and Peter Knirsch

Table of Contents

Applications

Theory

Working on OCL with graph transformations *

Paolo Bottoni[1], Manuel Koch[2],
Francesco Parisi-Presicce[1], Gabriele Taentzer[3]

[1] Università di Roma “La Sapienza Italy
[2] Free University of Berlin
[3] University of Paderborn

Abstract. The Object Constraint Language (OCL) provides an important complement to visual formalisms used in the definition of UML languages. Yet, its usage is limited by two major drawbacks. The first is the limited availability of tools for the automatic verification of constraints against model diagrams. The second is the difficulty of amalgamating a textual formalism such as OCL with the visual languages used in the rest of UML. We attack these problems with methods deriving from a graph-transformation based approach. We propose a visualisation of OCL, based on a recently proposed metamodel for it, which provides a declarative way to represent OCL constraints, and then we discuss how an operational semantics for OCL can be based on transformation units which guide the application of graph-transformation rules.

1 Introduction

The Object Constraint Language (OCL) provides an important complement to visual formalisms used in the definition of UML languages. It is used both as a textual counterpart to UML models, and to constrain extensions of UML using metamodels. Yet, its usage is limited by two major drawbacks. The first is the limited availability of tools for the automatic verification of constraints against model diagrams. The second is the difficulty of amalgamating a textual formalism such as OCL with the visual languages used in the rest of UML.

Both problems can be attacked based on the existence of some formal semantics for OCL. In particular, the existence of a metamodel, as introduced in [RG99], allows us to devise some form of visualisation, conforming to the metamodel, but which is better suited to integration with the UML diagrammatic languages. A visual formalism such as that of graph rewriting rules can also be used to provide an operational semantics for OCL, which can be applied both to its visual and to its textual representation. Such an operational semantics can be realised on top of existing tools for graph rewriting.

The paper first presents the main concepts behind the proposed visualisation and then discusses how an operational semantics can be achieved in terms of transformation units defining a strategy in the application of graph transformation rules. The conclusions illustrate some benefit of the approach

* Partially supported by the EC under Esprit Working Group APPLIGRAPH.

2 Visualisation of OCL Constraints

In the following, we consider two main issues when visualizing OCL. One is the visualization of navigation expressions, ubiquitous in UML. Another important issue is the visualization of collections and their operations. The visualization concepts for both issues are illustrated by examples taken from an industrial project on 'E-Government'. The project objective is to replace the existing software system used in the residents' offices in Berlin by a new software system that supports and facilitates both the business process within one or among several authorities and the business processes between the authority and the citizen by exploiting Internet technologies. The main responsibilities of the residents' registration office are the registration of inhabitants, the maintenance and preparation of the inhabitants data for other authorities like police or fire department, and the certification of passports and ID cards. The underlying business object model contains classes like `NaturalPerson` which describes natural persons as opposed to legal persons, and `Inhabitant` containing data of natural persons being inhabitants. Additional constraints must be checked to insure the consistency of the data base with the intended application. A more detailed presentation of the business model can be found in [BKPPT01]. In the same paper, the interested reader can also find a discussion on the principles behind the proposed visualization.

2.1 Constraints on Navigation Expressions

OCL constraints involve complex navigation expressions to reach object properties. A user trying to follow these expressions incurs in the cognitive cost of having to reformulate an object structure, given in some static structure diagram, in a different, textual, syntax. Hence, visualizing navigation paths would help the developer to maintain an overview of the structure while reasoning about the constraints. We propose to express object navigation in a visual, declarative way through collaboration diagrams.

The following is a simple OCL constraint stating that the birth date of a natural person comes before the date of moving into an apartment; the constraint is stated textually in the usual OCL syntax.

```
context NaturalPerson inv:
self.birth.attrBirthDate < self.address.attrDateOfMoveIn
```

The visualized form of this constraint, in Figure 1, contains three classifier roles, two of which present attributes. The attribute values, `x` and `y`, are compared in the bottom compartment. The kind of constraint is indicated by shortcuts 'inv', 'pre' or 'post' (for invariants, pre- or postconditions, respectively) in the upper left corner. Four alternative versions for visualizing this constraint could be employed, as navigation to `Birth` and `Address` objects can be performed on named as well as on anonymous associations.

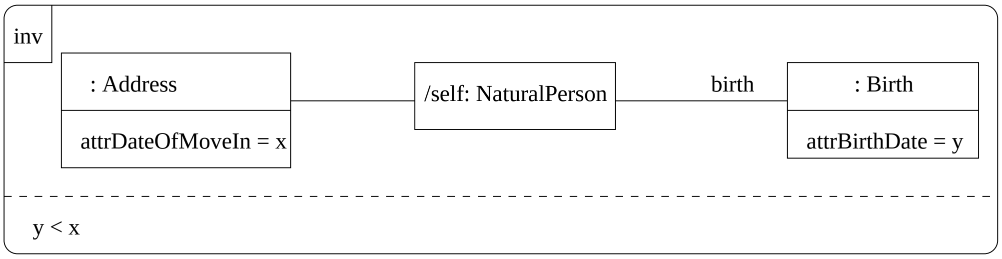


Fig. 1. *The birth date comes before the date of moving into an apartment.*

2.2 Collections and Their Operations

OCL supports three different types of collections: sets, bags and sequences. Sets are represented using the convention for multi-objects in UML. Under this convention, a multi-object represents all the objects, possibly an empty collection, reached by a navigation expression. We call this visual form *set box*. The other types of collections are represented by adorning the set box with details recalling the collection type, i.e. connecting the corners of the shifted rectangles with dots (to remind of a series) for a *sequence*, and placing a semi circle, reminding a handle, over the upper rectangle of the collection element for a *bag*. The concrete element type of a collection is in the front rectangle. Basing the visualization on collaborations, the application of an operation is described by an interaction.

We use the `select` operation to describe how a collection operation can be visualized. Usually, the selecting expression is framed by a set box. Such a representation is employed in Figure 2 to select all the addresses of a natural person who is a resident or a non-resident with a known address. The number of addresses must be greater than 0. The corresponding textual OCL constraint is:

context `NaturalPerson` inv: self.address→select(naturalPerson.inhabitant.attrState = #resident | #known)→ size > 0

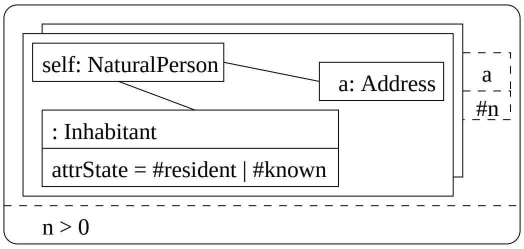


Fig. 2. A select statement

2.3 Logical expressions

The use of the alternative operator (|) in Figure 2 is a shortcut for a disjunction involving duplication of the classifier role to present the possible alternative values. A more general visualization has been devised for logical expressions. In particular, logical expressions on object navigation are represented by framing expressions in order to reproduce the nesting of logical AND and OR operators, where each level is alternately read as an AND or an OR, starting from an outermost AND frame. In case the original OCL formula had disjunctions at the top level, a new fictitious top node is first inserted to constitute the AND-labeled root, and then the translation process is started. Dashed diagram parts are used for negation.

If-then-else expressions can be depicted by frames with different compartments. The *if* compartment is above the *then* compartment on the left, and the *else* compartment on the right.

2.4 OCL Metamodel

As we base the visualization of OCL on collaborations, we perform some adaptation of the OCL meta model introduced in [RG99] and further elaborated in [Bod00] to make it consistent with the meta model for collaborations. The idea is to use collaborations to describe properties of objects. This is natural, since the description of object properties is based on classifier and association roles which are used to describe navigation paths. The dynamic aspects of collaborations are exploited to represent the calling of methods to determine object properties.

As in [Bod00], a special package UML_OCL contains basic data types and collections. Abstract collections are thought to be incorporated in this package as data types. They have to be instantiated by concrete element types, which is done in a special profile which introduces typed collections with a link back to classifiers to capture the collected type. Special OCL operations are integrated by offering special data types in the UML_OCL package as described above.

3 Graph Transformations for Checking Constraints

The main motivation to develop OCL has been the definition of well-formedness rules in the context of the UML semantics, but it may also be used for precise modeling of user applications. To express OCL semantics by graph transformation, a function $tr : Set(OclExpression) \rightarrow Rules \cup Set(TransformationUnit)$ is defined. An OCL constraint is an expression, with a boolean return value, satisfied by an instance model, if the corresponding rule or transformation unit can be applied to the instance graph. The evaluation of the rule or unit does not modify the graph on which the constraint is checked.

3.1 The Graph Transformation Approach

We work on directed, typed, and attributed graphs. Rule application follows the single-pushout approach to graph transformation [Löw93,EHK+96]. A rule may also contain a set of *negative application conditions (NAC)* to express that something *must not* exist for a rule to be applicable [HHT96]. The negative condition can refer to values of attributes [TFKV99]. Rules can also employ *set nodes*, which can be mapped to any number of nodes in the host graph, including zero. The matching of a set node is in any case exhaustive of all the nodes in the host graph satisfying the condition indicated by the rule. Set nodes have to be preserved between the left and right context, but new set nodes can be created. This implies that it is not possible to use a rule to destroy a set of nodes, but the nodes in the set have to be removed individually from it. Finally, set nodes must not occur in NAC's.

Transformation units are used to further control rule application. In the following example, the denotation of transformation units is mainly reduced to the containing rules and control conditions. Initial and terminal configurations are all instances of the given UML model. The import relation of units remains implicit. The control condition is specified by expressions over rules.

Given a set $Names$ of rule names from which rule expressions are constructed, a rule expression E as we will use it, is a term generated by the following syntax:

- basic operators:
  $E ::= Names \mid E_1$ **and** $E_2 \mid E_1$ **or** $E_2 \mid E_1; E_2 \mid \mathbf{a}(E) \mid \mathbf{na}(E) \mid$
  null | **if** E_1 **then** E_2 **else** E_3 **end** | **while** E_1 **do** E_2 **end**
- derived operators:
  $E ::= E_1$ **implies** $E_2 \mid E_1 \ = \ E_2 \mid E_1$ **xor** E_2 | **asLongAsPossible** E **end**

Most of the operators presented above have the obvious meaning: operators **a** and **na** test the applicability and non-applicability, resp. Each rule expression is either applicable or non-applicable, i.e. it has a boolean return value. In case, it is applicable, the unit it controls can also produce a value, a node, or a set node, according to the declaration of the arguments for the unit. In the operators **if-then-else** and **while-do-end**, the rule expression E_1 is tested for applicability (without being applied). The result of this test determines how to proceed with application in the usual way. Operator **asLongasPossible** applies a rule expression to a graph as long as it is applicable. A detailed formal definition of rule expressions as presented above can be found in [BKPPT00].

3.2 Checking simple OCL constraints

An OCL constraint over navigation expressions can be translated into a graph rule, easily derivable from the visualization of such OCL constraints by static collaboration diagrams. Such diagrams can be interpreted as identical graph rules, i.e. both sides are equal and the rule morphism is the identity (e.g. the OCL constraint in Figure 1 which can be interpreted as an identical graph rule).

If a rule which is the translation of a constraint can be applied to some instance graph, the constraint is satisfied for this instance. The non-applicability of a rule can have two causes: either there is no total mapping of the left-hand side to the instance graph or no mapping satisfies all the additional application conditions. In both cases, this can be reported to the user helping him/her to find the inconsistency. After the user has performed further editing steps on the UML model, the checking can be started again. Thus, the rule-based character of inconsistency checking could be advantageously used to perform constraint checking any time the user so wishes.

3.3 Checking advanced OCL constraints

The proposed visualisation does not always have a direct procedural counterpart and direct matching is not sufficient when some strategy has to be followed in constraint checking. For example, checking a constraint expressed as a nesting of logical connectives, or navigation expressions involving a universal quantifier requires following a precise sequence of actions.

The translation from OCL constraints to rule expressions can be performed systematically, as the translation to the declarative visualisation. We consider here again the example in Figure 2. We assume that the predefined operations, such as `select`, have already been translated into suitable transformation units. Then, the whole constraint is translated again into a transformation unit which uses the select unit.

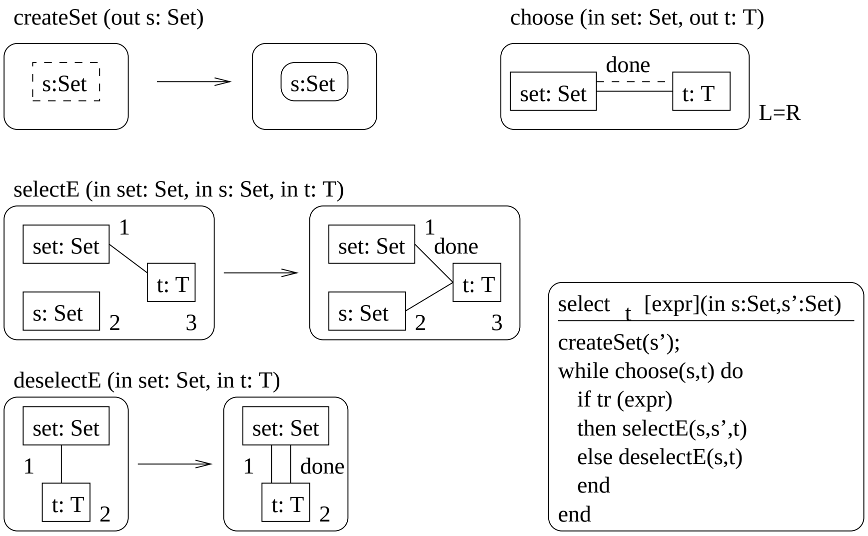


Fig. 3. Translation of pre-defined operation `select`

The `select` operation is directly dependent on the evaluation of the selecting expression. The selecting expression is part of the resulting transformation unit, thus the unit name is dependent of the expression. (Consider the translation of `select` in Figure 3.) In the body of this transformation unit, the operation is first initialized by creating an additional structure (here a `Set` node) and then the select action is performed. Note that the dotted box in the left-hand side of rule `createSet` indicates that the set s must not be already present in the diagram for the rule to be applicable. For each element chosen from the given set (and not already visited, i.e. without an adjacent `done` edge), we have to select or deselect it, depending on the selecting expression. The chosen element is used as input parameter of the selecting expression.

Having the translation of `select` available, the translation of the OCL constraint in Figure 2 is depicted in Figure 4. The first rule navigates to all the addresses. Then two transformation units are called, performing the selection of addresses of residents or known persons and counting them. The last rule checks if there is at least one of those addresses.

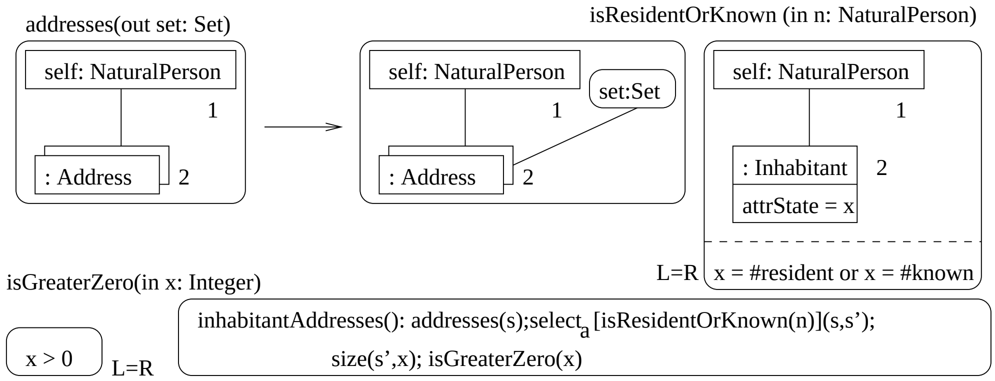


Fig. 4. Translation of OCL constraint in Figure 2

During a constraint check, the instance graph may be augmented by additional objects and links. These objects, which may be collection nodes and additional `done` edges, have to be deleted by additional rules collected in a special transformation unit which is invoked after each check.

3.4 An example on an instance diagram

As an example of consistency check of an instance diagram, consider the object diagram in Figure 5, showing a portion of the data base in the E-Government project. We want to check the OCL constraint in Figure 2 on the two instances of `NaturalPerson` there.

To test the well-formedness of a diagram or model, we have to look for the applicability of a set of rules and/or transformation units. Looking at our sample model in Fig. 5, the transformation unit in Fig. 4 can be applied fully to the portion of the diagram relative to Manuel, but not to the portion relative to Gabi. Indeed, the transformation unit in Figure 4 will start by constructing the set of addresses, which results empty for Gabi. Now the `select` transformation unit will start by constructing the new set to accommodate the selected address, but it will not enter the `while` loop, as the rule defining the looping condition is not applicable. Hence, the `select` transformation unit will end by returning

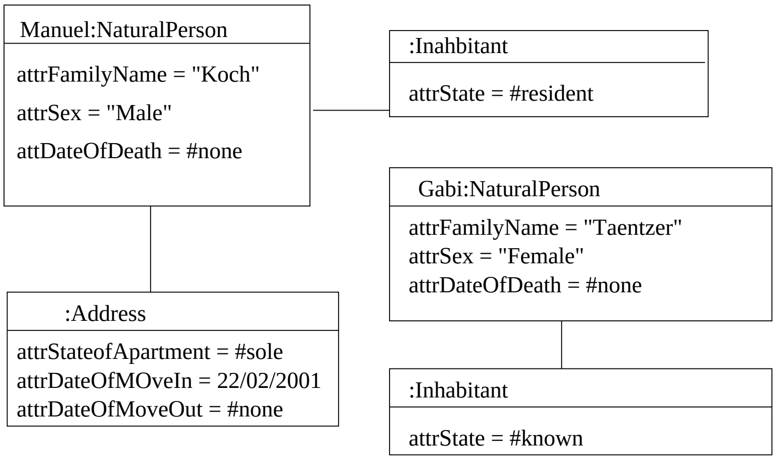


Fig. 5. An object diagram not satisfying a constraint

an empty set. The `inhabitantAddresses` transformation unit will then proceed to compute the size of this set and finally the rule for `isGreaterZero` is not applicable, making the whole unit to fail. Conversely, when applying the unit to Manuel's portion, the selected set contains the only address present for Manuel, so that it is not empty, and its size is greater than zero, testifying to that the constraint is satisfied.

3.5 Discussion

The identification of the transformation units translating a textual OCL constraint can proceed in parallel with the translation to a visual counterpart, providing an operational reading of the declarative visual representation. The two visual forms of management of OCL constraints, i.e. visual OCL constraints and graph transformations, share some commonalities at the base level, presenting elements from the UML syntax, namely, classifiers, associations, multi-objects and attributes. However, the composition of complex constraints is expressed through ad hoc visualisations in one case and through rule sequences in the other. In general, rule expressions can be employed to govern the application of rules for complex constraints, in a way which can be inferred by the structure of the visualisation. Conversely, the visualisation provides suitable graphical constructs to express the iteration and alternative operators of rule sequences.

Using graph transformation to give OCL an operational semantics is rather closely related to the dynamic metamodeling approach by Engels *et al.* who provide a graphical approach to the operational semantics of behavioural UML diagrams [EHHS00]. To this end, they use collaboration diagrams, which can be interpreted as graph rules. The two approaches mainly differ in the way rule application is controlled. While in the metamodeling approach rule applications can be controlled by sequential and parallel composition as well as the usage of other transformation units, we allow a larger variety of control constructs.

4 Conclusions

We have proposed to exploit forms of visualisation to achieve a smoother integration and use of OCL in the context of the UML diagrammatic languages, both from a declarative and from a procedural point of view. To this end the paper has illustrated how visual representation of constraints exploiting the UML visual syntax can be achieved, and how graph rules can be used to support consistency checking of OCL constraints on target diagrams. Two systematic translations from the textual OCL syntax to the two forms of visual syntax (static visualization and executable graph transformation units) preserving the semantics of the constraints can be realized. These visual syntaxes admit some amount of hybridization with textual syntaxes, as conditions on properties or primitive OCL operators (such as `size` or `isOCLKindOf`) are more simply left in the textual form (see for instance the realization of isGreaterZero in Figure 4). The presented visualisation is based on collaborations, and is consistent with the metamodel for OCL proposed in[RG99]. It introduces a limited amount of new core notation, but offers a variety of visual shortcuts for convenient visual notation, favouring a greater readability and amenability to reasoning of OCL constraints. The combination of the visualisation of OCL and the application of rule expressions has the advantage of allowing an intuitive representation to the user, who can perform direct checking on the model, and of getting a formal semantics, in terms of transformation units.

Based on the proposed translation, an OCL evaluator can be implemented on top of a graph transformation machine like AGG or PROGRES ([EEKR99]) and later integrated into a UML CASE tool. These tools support a step by step evolution of the underlying host graphs. An OCL evaluator based on such a graph transformation machine can help to understand the implemented OCL semantics by following the stepwise evaluation on instance diagrams visually. An editor for visual OCL constraints is currently being implemented for the open source CASE tool ArgoUML. The graph transformation-based approach to checking inconsistencies can easily support automatic repair actions, by defining suitable graph rules to solve them, if possible. This approach relies on the idea of living with inconsistencies during software development presented in [GMT99], also on the basis of graph transformation.

References

[BKPPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans and S. Kent, editors, *UML 2000*, number 1939 in LNCS, pages 294–308. Springer, 2000.

[BKPPT01] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. In M. Gogolla and C. Kobrin, editors, *UML 2001*, number 1939 in LNCS, pages 257–271. Springer, 2001.

[Bod00] M. Bodenmüller. The OCL Metamodel and the UML-OCL package. Proc. of OCL Workshop, Satellite Event of UML 2000, York, October 2000, 2000.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools.* World Scientific, 1999.

[EHHS00] G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In A. Evans and S. Kent, editors, *UML 2000*, number 1939 in LNCS, pages 323–337. Springer, 2000.

[EHK+96] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 247–312. World Scientific, 1996.

[GMT99] M. Goedicke, T. Meyer, and G. Taentzer. ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proc. 4th IEEE Int. Symposium on Requirements Engineering (RE'99), June 7-11, 1999, University of Limerick, Ireland.* IEEE Computer Society, 1999. ISBN 0-7695-0188-5.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

[RG99] M. Richters and M. Gogolla. A metamodel for OCL. In R. France and B. Rumpe, editors, *UML'99*, pages 156–171. Springer LNCS 1723, 1999.

[TFKV99] G. Taentzer, I. Fischer, M. Koch, and V. Volle. Visual Design of Distributed Systems by Graph Transformation. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*, pages 269–340. World Scientific, 1999.

Towards Automatic Translation of UML Models into Semantic Domains

Reiko Heckel, Jochen Küster, Gabriele Taentzer

University of Paderborn, Germany
{reiko,jkuester,gabi}@upb.de

Abstract. The use of UML for software specification leads usually to lots of diagrams showing different aspects and components of the software system in several views. In order to support a view-oriented approach to system modeling, consistency in views and in between views has to be manageable. It is a reasonable approach to consistency management when first choosing a suitable semantic domain, provide a partial mapping into this domain, and specify as well as verify consistency constraints formulated in that domain. Annotated meta model rules can be used to translate elements of UML models into the semantic domain chosen. In this contribution, we consider triple graph grammars and attributed graph transformation approaches for the precise definition of meta model rules and outline the tool support for automatic translation.

1 Motivation

As there is currently no formally defined semantics for UML, consistency checking of UML models on a semantic basis cannot be done within the UML itself. In principle, there are at least two different ways of tackling the problem of achieving semantic consistency: One is to define a complete formal semantics for the UML. The other is to establish semantic consistency using an approach not relying on such a complete semantics.

Our approach for consistency checking relies on the observation that semantic consistency checking can also be performed by choosing an appropriate semantic domain for a particular consistency problem [EKGH01]. A semantic domain has to be appropriate in the sense that it must allow the formal specification and (automatic) verification of consistency constraints of the UML models. For different consistency problems also different semantic domains can be chosen. As a consequence, the specification of mappings of UML models into various semantic domains is of major interest because such a mapping is essential for checking consistency of UML models following our approach.

In order to illustrate our approach, we introduce the running example of a statechart modeling the behavior of a person. In the UML, a statechart can be associated to a class in order to specify the *object life cycle*, i.e., the order of operations called upon an object of this class during its life-time. Given a class A and a subclass B of A, the behavioral conformity of the associated statecharts gives rise to the problem of statechart inheritance. In the literature, different notions are proposed (see, e.g., [EE95,EE94,HK99]).

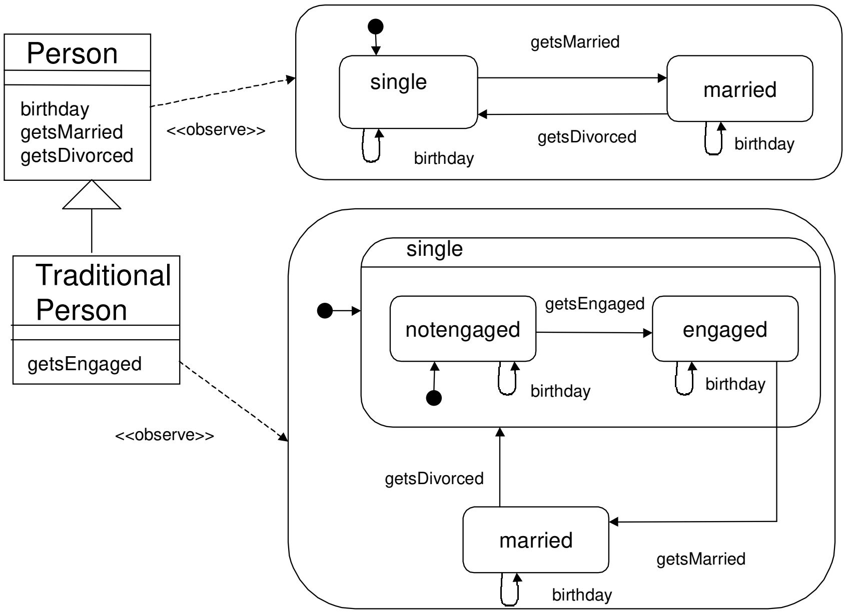


Fig. 1. The Person example

For analyzing consistency of such models, we have proposed a mapping of statecharts into CSP [EKH01]. Communicating Sequential Processes (CSP) [Hoa85] provide a mathematical model for concurrency based on a simple programming notation and supported by tools [For97]. In fact, the existence of *language* and *tool support* are most important to our aim of *specifying* and *verifying* consistency constraints. The semantics of CSP is usually defined in terms of *traces*, *failures*, and *divergences* [Hoa85]. Together with these semantic models come several notions of process refinement which can then be used for specifying consistency constraints between processes obtained from the translation of statecharts to CSP. For example, trace refinement between the process of a superclass and a subclass can be used for establishing specific notions of behavioral conformity.

The mapping of statecharts into CSP can be described by specifying mapping rules using extracts from the metamodel of UML diagrams. The metamodel for statecharts given in Figure2 defines a simplified notion of a statechart. The presentation conforms to the UML meta model but for the flattening of some inheritance relations. All meta classes contain a meta attribute name:string which is not shown in the figure. (In the UML meta model this is inherited from the super class ModelElement).

Mapping rules for statecharts to CSP can then be described by the rules in Figures 3 and 4 combining textual grammar rules with graphical patterns. Consider, for example, rule (2) in Figure 3 which defines the semantics of a composite (OR) state in terms of the semantics of its default state. In the center, this is represented by a meta model pattern showing the abstract syntax of the source language UML. On the right, a rule is shown that generates the

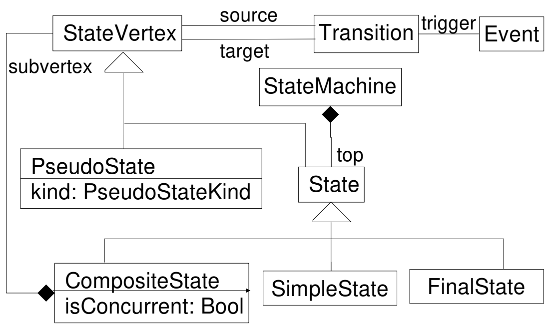


Fig. 2. UML meta model fragment for protocol statecharts

corresponding expression in the target language CSP. It is parameterized over the names of the UML model elements and contains non-terminals like `extBeh` which are to be replaced by application of other rules. On the left, the corresponding concrete UML syntax is shown. The result of the application of the rules to the example of the Person example can be found in [EKH01].

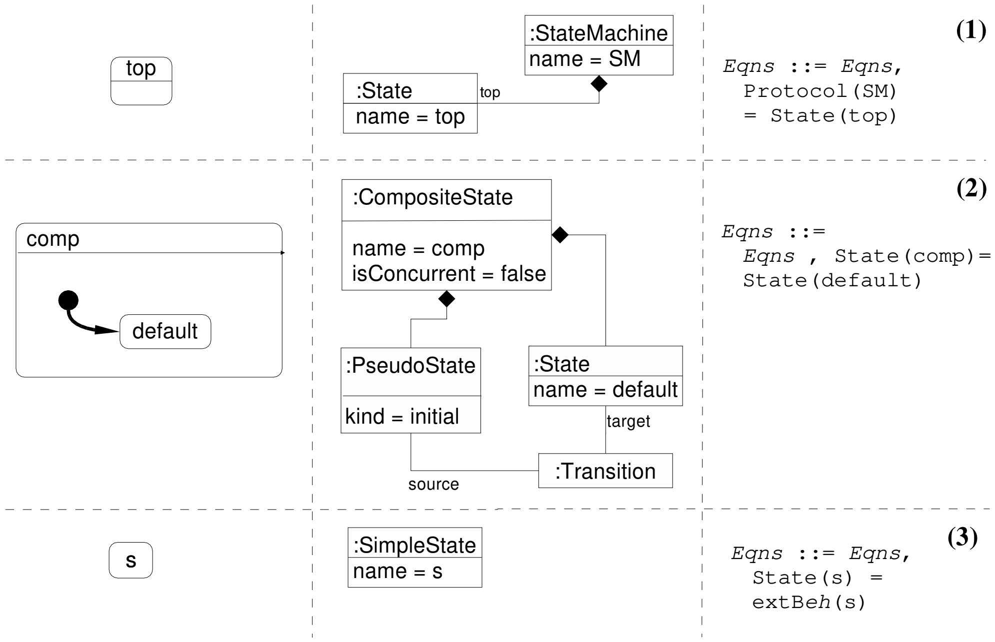


Fig. 3. Mapping rules for states

These mapping rules describe the translation into a semantic domain rather precisely, but still informally. So, what are the possibilities to formalize, and moreover, implement such a mapping? In the following, we discuss two approaches: The formalization by attributed graph transformation and its im-

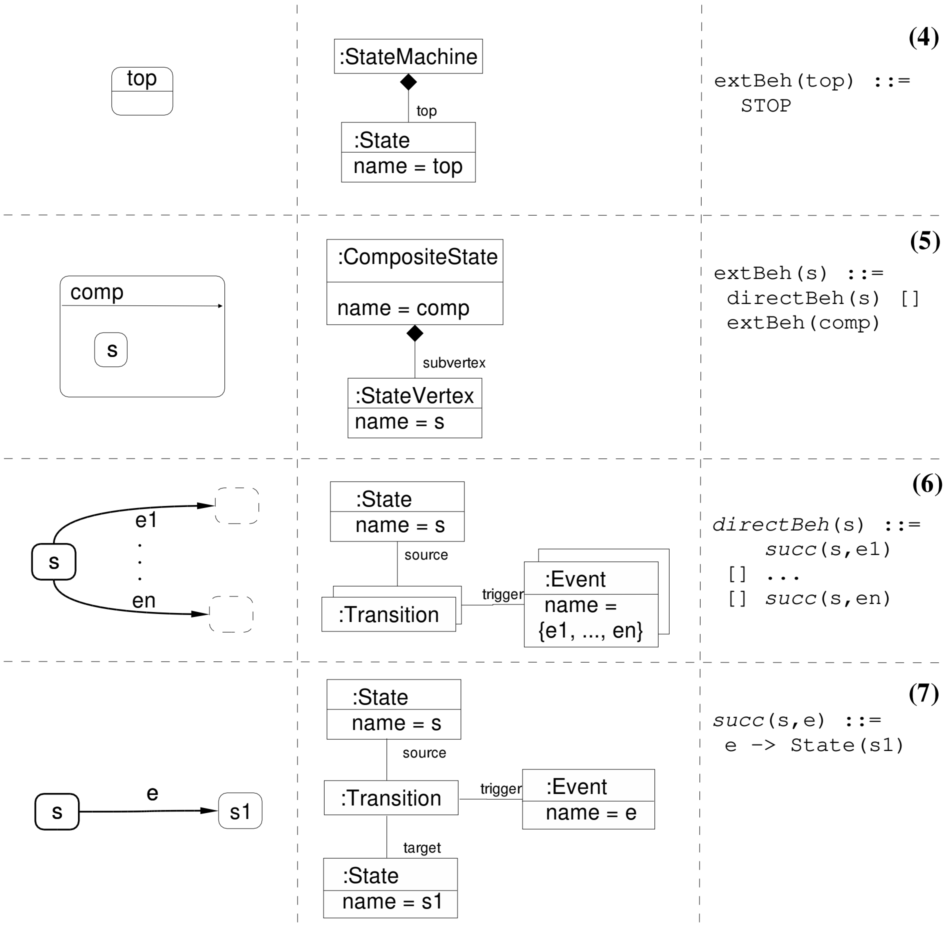


Fig. 4. Mapping rules for the behaviour

plementation by AGG and a realization by pair/triple graph grammars. Both approaches are roughly compared in the conclusion.

2 Translation by Attributed Graph Transformation

Mapping rules as presented in the previous section can be formalized by attributed graph transformation. We will discuss in the following how mapping rules can be systematically translated into attributed graph rules. Using AGG as an engine for attributed graph transformation it becomes possible to automatically translate UML models into some semantic domain and to do semantic consistency checking by further tools. For the whole procedure we assume that the UML model is already checked to be syntactically correct.

Semantic attributes. In the first step, we extend the metamodel representation which shows the abstract syntax of a UML model by attributes containing semantic expressions. Each model element which shall be equipped with corresponding semantic expressions is extended by further attributes containing some semantic information. Dependent of the semantic domain the attribute type can be string or graph or a more specific type. This semantic extension of abstract syntax graphs can be seen in the very analogy to attributed syntax trees used for textual languages. Attributed graph transformation systems for graph languages can be used analogously to attribute grammars for textual languages ([Paa95]). In particular, there might be an analogous need for inherite d or derived attributes for languages where graphs contain trees as substructures.

Mapping rules as presented in the previous section can be systematically translated into attributed graph rules as follows: The attributed graph rules needed for language translation build upon identical pattern rules $id_p : P \to P$ dealing with meta model extracts. Since attributed graph transformation as implemented in AGG cannot handle inheritance of node or edge types, the type information has to be flattened. Additional attributes can be used to further specify subtypes. The translation to target expressions is done in attribute computations. Testing on the left-hand side that certain semantic attribute values are set and additional attribute conditions are fulfilled, the graph transformation computes new values for semantic attributes. An attribute condition is used to test if the left-hand side of a target rule can match.

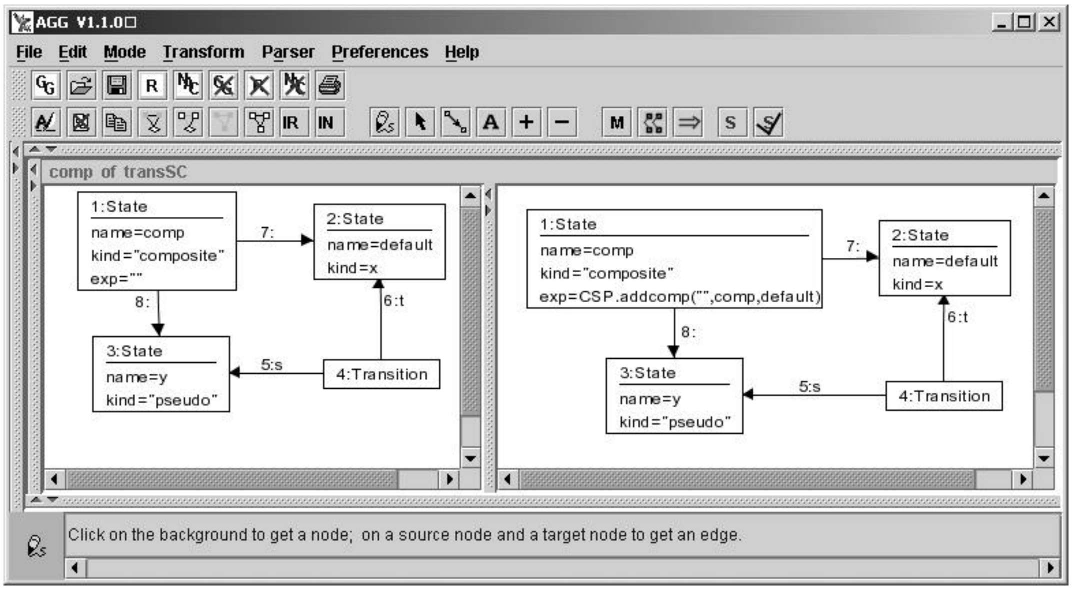


Fig. 5. Mapping rule (2) as attributed graph rule

Application to sample mapping rules. Now we consider how the semantic attributes in our example would look like. Mapping statecharts to CSP a new equation is created for the state machine and for each state not being a pseudo state. Thus, nodes of type `StateMachine` and `State` get a semantic attribute `exp` each containing the corresponding CSP equation after translation. Moreover, we need a semantic attribute `extBeh`, since the external behaviour is inherited from the superstate. The graph part of the attributed translation rules consists of identical pattern rules for the meta model pattern shown in Figures 3 and 4 in the previous section. Consider e.g. the sample rules (2) and (5) in Figures 5 and 6.

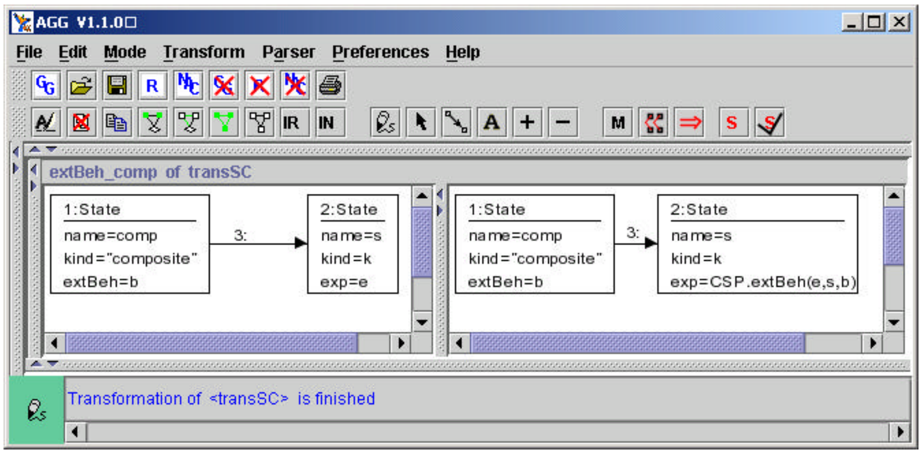


Fig. 6. Mapping rule (5) as attributed graph rule

In rules (1) - (3) the attribute computation is rather easy. After testing that the semantic expression is not yet computed the rules just create CSP equations for the state machine and its states. Rules (4) - (7) contain the context-free replacement of non-terminals of the target language. Thus, we have to test the occurrence of non-terminals on the left-hand side and replace this occurrence by the target's right-hand side expression.

AGG [ERT99] is a graph transformation engine that allows to attribute graph objects by Java expressions. We use AGG to formulate the sample mapping rules by attributed graph rules. While the graph part description is straight forward we focus on the attribute computation in the following. To apply rules (4) - (7) the existence of non-terminals has to be tested. E.g. for mapping rule (5) attribute conditions

– `e.indexOf(``extBeh(``+s+'')'') != -1` and
– `!k.equals(``pseudo'')`

are added. The proper attribute computation is encapsulated in a separate Java class. Each translation rule refers to a method of this class which computes the new CSP expression. Since the target language is rather simple we implemented its context-free replacement rules directly in Java. Consider the following section of class `CSP` to understand how the computation can look like. For complex target languages the usage of a parser generator like JavaCC ([Jav01]) can be helpful.

```
 public class CSP {
    ...
    public static String addcomp(String exp, String
comp, String name){
       String s = "State("+comp+") = State("+name+");";
       exp = exp + s;

     return exp;
     }

    public static String extBeh(String exp, String name,String b) {

 String searched = "extBeh("+name+")";
       int l = searched.length();
       int i =
exp.indexOf(searched);
       String pre = exp.substring(0,i-1);
       String post =
exp.substring(i+l);
       if (name.equals("top"))
           exp = pre + "STOP" + post;
      else exp = pre + " directBeh("+name+ ") [] " + b + post;
      return exp;
     }
   ...
}
```

Mapping rule (6) in Figure 4 contains multi objects which would be formalized by rule schemes in attributed graph transformation for any number of transitions and events. Since AGG not yet supports rule schemes rule (6) has to translated into three rules iterating through the set of event objects. The full AGG grammar for the translation of our sample statechart in Figure 1 to CSP can be found at http://tfs.cs.tu-berlin.de/agg/examples/statecharts. Since this grammar is conflict-free, i.e. no two rule applications are parallel dependent, the translation does not need further application control. It leads always to a unique result without backtracking. (See also the work on reserved graph grammars. [ZZ97])

Tool support. To use AGG in a larger tool chain translating (parts of) UML models to semantic domains and analyze them, the following steps have to be performed: A UML model is defined in any CASE tool that supports the storage of UML models in the XML Metadata Interchange (XMI) format ([XMI01]). Then, the Extensible Stylesheet Language (XSL) [W3C01] can be used to transform a UML model in XMI format into a graph in GGX format which is the input format for AGG. Applying the translation rules described above as long as possible to this graph creates the semantic expressions. Stored in a file they provide the input to further tools offering semantic analysis. E.g. translating statecharts into CSP expressions offers the possibility to check consistency by a model checker like the FDR tool (mentioned in Section 1).

3 Translation by Pair or Triple Graph Grammars

In this section, we shall briefly review two alternative approaches to the specification of translators between graphical and textual languages and discuss their applicability to our problem. Both pair- and triple graph grammars are based on a coupling of the production rules for the source and target language, which allows derivations in the source language to be translated into derivations of the target language. Both approaches are symmetric, that is, the role of source and target grammars can be exchanged.

Pair and triple graph grammars. Pair grammars [Pra71] are based on context-free graph grammars of which context-free Chomsky grammars form a special case. This restriction is motivated by the fact that the translation of a given source structure involves the problem of finding a derivation of this structure using the source grammar, that is, an efficient parser is required. Unfortunately, most interesting diagram languages, like statecharts, class diagrams, etc. have an arbitrarily complex graphical structure, that is, they are not context-free.

This is one of the motivations for triple graph grammars [Sch94] which allow for non-context-free source and target graph grammars.[1] In fact, in the meantime, efficient graph parsers are available also for more general classes of graph languages. In particular, in the case that the graph grammars are monotonic, the triple graph grammar approach can be used to generate graph translators automatically from the mapping specification. The overall strategy is still the same, that is, parsing the source graph and playing the resulting derivation back to construct the target graph.

Application to sample mapping rules. Let us try to understand this in terms of our example, the mapping of statechart diagrams to CSP. The mapping rules in Figures 3 and 4 consist of two parts: a pattern for the source graph (the meta model presentation of a statechart diagram) and a context-free production for generating CSP expressions. The relation between the two parts is implicit in the names of the states, which are used as parameters of process names. In order to view such a mapping rule as a triple (or pair) grammar rule, we could follow the idea of the previous section considering the meta model pattern as an idle production (where left- and right-hand side coincide). However, since according to the application strategy described above, the source productions should be used to generate statechart diagrams, such idle productions are not very useful.

Rather, we have to designate for each pattern P the left-hand side L of the corresponding source production $p : L \hookrightarrow P$. The resulting source grammar

[1] The name *triple graph grammar* refers to a third grammar in between source and target which produces a mapping structure to keep track of the relation between the source and target structures.

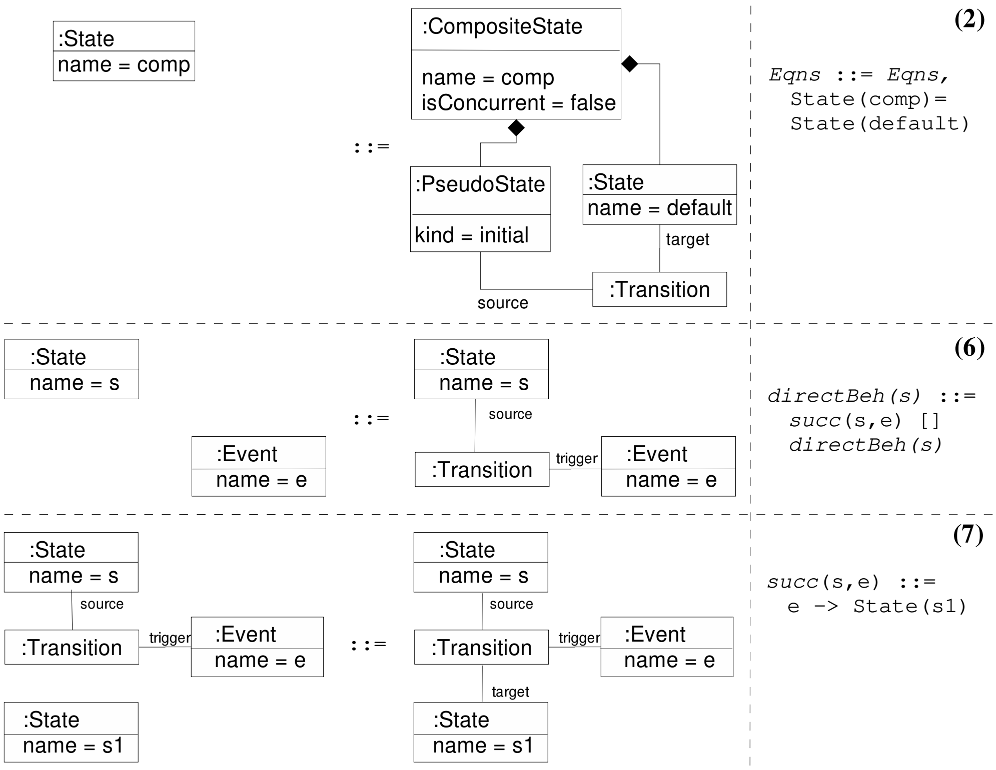


Fig. 7. Generalized pair grammar rules for mapping rules (2) and (7)

Src should then generate a superset of the source language, that is, we assume that we are already given syntactically legal source graphs. In our example, this would mean that the triple/pair grammar presentation of the rules (1) and (2) have empty left-hand sides. The presentation of rule (2), shown in the upper left of Figure 7, promotes a State instance to a CompositeState instance. This does not violate the monotonicity of the rule because CompositeState is a subclass of State, i.e., migrating to the latter means to add additional structure while preserving the identity of the object. The production rule for (6) in Figure 7 selects a source state and an event and adds a “dangling transition” which is completed by rule (7) by the target state. The production rule (5) generates a subvertex link between a given CompositeState and StateVertex. This creates a potential conflict with rule (2) which also generates subvertex links. A deeper analysis of the structure of statecharts shows that we can omit in rule (2) the subvertex link between the default State and the CompositeState, while an application of rule (5) to pseudo states is not needed.

The target grammar productions shown on the right are already context-free and can thus be left untouched. The resulting rules constitute a mix of pair and triple grammars, with monotonic graph production for the source and context-free Chomsky grammar productions for the target structure.

4 Conclusion

Motivated by the need to translate UML models into semantic domains, we have outlined two approaches to formalize mapping rules from graphical into textual languages. This special case of mapping graphical languages is particularly interesting because most formal methods and programming languages are based on textual syntax. Next, the two approaches shall be compared w.r.t. their structure of rules, their application strategy, the tool support required, and the additional effort necessary for converting informally stated rules like the ones in Figure 3 and 4 into the respective rule format.

With respect to rule structure, attributed graph rules are apparently simpler than pair/triple graph grammar rules because the latter consist of two or three separate (but connected) rules. However, if we regard the attribute computation of an attributed graph rule as a separate part, we obtain a similar structure. The structural difference is, in fact, a matter of syntax since both rule formats consist of a part working on the source structure (graph transformation rules in both cases), a part generating the target structure (attribute expressions in attributed grammars and textual rewriting rules in our version of pair/triple graph grammars), and a (more or less explicit) connection between the two.

The application strategies of the two approaches are fundamentally different. Attributed graph rules have to be designed in such a way that the resulting rewrite system is confluent so that every computation leads to the desired result. Triple graph grammars are usually applied by first parsing the source structure and then using the resulting derivation for controlling the creation of the target structure. Here, more flexibility is given by the potential non-determinism of the parsing process.

This difference results in different requirements for tools implementing the two approaches. While attributed graph grammars can be evaluated by nearly all implementations of graph transformation accommodating attributes, the pair/triple graph grammar approach requires more specialized tools, like a graph parser and a mechanism for controlling the generation process.

Most interesting for the usability of the approach is the question, how big is the effort of converting the informal mapping rules into rules following one or the other approach. In both cases, this requires additional information, like the distinction between preserved and to-be-generated items in source rules of pair/triple graph grammars, or the association of semantic attributes to model elements in the case of attributed grammars. The impression is that the latter can be done in a more systematic way, whereas the design of the graph part of the pair/triple graph grammar rules requires more insights in the structure of the problem: Different parsing grammars for statecharts may exist which leads to the problem of deciding which one to choose and how to adapt the right rules to fit to the parsing grammar.

Altogether, we feel that the attribute rules are closer to the original intuition of our mapping rules as declarative specification of meta model mappings, while triple graph grammars force us to think in terms of a generation process for the source structure. In particular, the application of triple graph grammars forces a complete parsing of the structure even to provide only a partial translation of the model into a semantic domain. Considering e. g. the translation of timing constraints of a sequence diagram into a system of linear inequalities, the application of triple graph grammars therefore seems to be rather inefficient. A more formal comparison of the approaches and their expressiveness will be subject to future work.

References

[EE94] J. Ebert and G. Engels. Structural and behavioral views of OMT-classes. In E. Bertino and S. Urban, editors, *Proceedings, Object-Oriented Methodologies and Systems*, LNCS 858, pages 142–157. Springer-Verlag, 1994.

[EE95] J. Ebert and G. Engels. Specification of Object Life Cycle Definitions. Fachberichte Informatik 19–95, Universität Koblenz-Landau, 1995.

[EKGH01] G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC)*, pages 186–195. ACM Press, 2001.

[EKH01] G. Engels, J. M. Küster, and R. Heckel. Rule-based specification of behavioral consistency based on the uml meta-model. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools., 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, LNCS 2185, pages 272–287. Springer, 2001.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.

[For97] Formal Systems Europe (Ltd). *FDR2 User Manual*, 1997.

[HK99] D. Harel and O. Kupferman. On the Inheritance of State-Based Object Behavior. Technical Report MCS99-12, Weizmann Institute of Science, Faculty of Mathematics and Computer Science, June 1999.

[Hoa85] C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.

[Jav01] Java Compiler Compiler - JavaCC. http://www.webgain.com/products/java_cc/, 2001.

[Paa95] J. Paakki. Attribute grammar paradigms, a high-level methodology in language implementation. *ACM Surveys*, 27(2):196 – 255, 1995.

[Pra71] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.

[Sch94] A. Schürr. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 903, pages 151–163. Springer-Verlag, 1994.

[W3C01] W3C. The Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/, 2001.

[XMI01] XMI - XML Metadata Interchange. http://www.oasis-open.org/cover/xmi.html, 2001.

[ZZ97] D.-Q. Zhang and K. Zhang. Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs. In *Proc. IEEE Symposium on Visual Languages (VL'97)*, pages 288–295, Los Alamitos, CA, 1997. IEEE Computer Society Press.

Processing Causal Block Diagrams with Graph Grammars in AToM^3

Ernesto Posse[1], Juan de Lara[1,2], and Hans Vangheluwe[1]

[1] School of Computer Science
McGill University, Montréal
Québec, Canada
eposse@cs.mcgill.ca, hv@cs.mcgill.ca
[2] ETS Informática
Universidad Autónoma de Madrid
Madrid, Spain,
Juan.Lara@ii.uam.es

Abstract. AToM^3 is a tool which supports *multi-formalism* modelling and *meta-modelling* to facilitate computer assisted analysis and design of complex systems. To enable the automatic generation of modelling tools, the formalisms themselves are modelled at a meta-level within an appropriate meta-formalism. The generated tools are able to process (create, edit, simulate,...) models expressed in the corresponding formalism. AToM^3 relies on graph grammars and graph rewriting techniques to perform the transformations between formalisms as well as for other tasks, such as code generation, model optimization and simulator specification. As a case study, we describe the syntax and operational semantics of Causal Block Diagrams (CBD). The animation of such operational semantics results in the actual simulation.

Keywords: Modelling & Simulation, Meta-Modelling, Multi-Formalism Modelling, Graph Grammars, Operational Semantics.

1 Introduction

AToM^3 is a visual Meta-Modelling tool developed by the authors, which supports modelling of complex systems. Complex systems are characterized by components and aspects which, in addition to being numerous, have structure and behaviour which cannot be appropriately described in a single formalism. Examples of commonly used modelling formalisms are Differential-Algebraic Equations (DAE), Causal Block Diagrams, Petri Nets, Entity-Relationship diagrams (ERD), and State Charts.

From the meta-specification of a modelling formalism, AToM^3 is able to produce customized tools to process models specified in the described formalism. Both syntax and semantics of a formalism are modelled. Some of the model manipulations in which we are interested include transformations to other formalisms, simulations, optimizations and (textual) code generation for other tools.

Causal Block Diagrams (CBD) are a general formalism used for modelling of causal, continuous-time systems. The simulation of such systems on digital computers requires a discrete-time approximation. There are several approaches to this simulation problem. One interesting solution is to describe CBD syntax in an appropriate CBD meta-model and to provide a specification of the operational semantics of such diagrams using graph grammars. The animation of such operational semantics will result in the actual simulation. We can thus regard the graph grammar as an *executable specification*. This approach is desirable for its generality, since it can be applied to a wide class of formalisms besides CBD. There is, however, a tradeoff made between generality and efficiency. As a general rule, customized, hand-coded, formalism-specific simulation algorithms are more efficient. The approach of relying on graph grammars is expensive due to the nature of the graph matching algorithm. However, there are other motivations, both theoretical and practical:

- Explicitly defining the operational semantics of any formalism should be considered as part of the design of the actual simulator, providing a specification, from which a more efficient implementation could be built.
- The specification also provides a framework (a reference implementation) for verifying and testing different implementations.
- It provides a portable simulator, since it is more abstract than a hand-coded implementation.
- It allows for reasoning about the described systems. For example, it allows for the definition of general algorithms for bisimulation.

The rest of the paper is organized as follows. Section 2 describes the motivations for meta-modelling. Section 3 relates graph-grammars to meta-modelling. Section 4 gives a brief description of AToM^3's architecture. Section 5 describes the specification (meta-model) of the CBD formalism. Section 6 provides the definition of CBD's semantics in terms of graph grammars.

2 Meta-Modelling

One of the characteristics of complex systems is the diversity of their components. Consequently, it is often desirable to model the different components using different modelling formalisms. This is certainly the case when inter-disciplinary teams collaborate on the development of a single system. Flexibility is also required as different teams may prefer slight variations of a particular formalism. A proven method to achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the modelling language itself [4][10]. Such a model of the modelling language is called a meta-model. It describes the possible structures which can be expressed in the language. A meta-model can easily be tailored to specific needs of particular domains. This requires the meta-model modelling formalism to be rich enough to support the constructs needed to define a modelling language. Taking the methodology one step further, the meta-modelling formalism itself may be modelled by means of a meta-meta-model. This meta-meta-model specification captures the basic elements needed to design a formalism. Table 1 depicts the levels considered in our meta-modelling approach.

Level	**Description**	**Example**
Meta-Meta-Model	Model describes a formalism that will be used to describe other formalisms.	Description of Entity-Relationship Diagrams, UML class Diagrams
Meta-Model	Model describes a simulation formalism. Specified under the rules of a certain Meta-Meta-Model	Description of Deterministic Finite Automata, Ordinary differential equations (ODE)
Model	Description of an object. Specified under the rules of a certain Meta-Model	$f'(x) = -\sin x, f(0) = 0$ (in the ODE formalism)

Table 1. Meta-modelling levels.

Formalisms such as ERD are often used for meta-modelling. To be able to fully specify modelling formalisms, the meta-level formalism may have to be extended with the ability to express constraints (limiting the number of meaningful models). For example, when modelling a Determinsitic Finite Automaton, different transitions leaving a given state must have different labels. This cannot be expressed within the ERD formalism alone. Expressing constraints is most elegantly done by adding a constraint language to the meta-modelling formalism. Whereas the meta-modelling formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems [6], including AToM^3 use the Object Constraint Language OCL [8] used in the UML.

Fig. 1 depicts the structure we propose for a meta-modelling environment. AToM^3 was initialized using a hand-coded ERD meta-meta-model. As the ERD formalism can be described as an ERD model, the environment was subsequently bootstrapped. Meta-formalisms are described by meta-meta-models. Although it is possible to describe a meta-formalism mf_1 using another meta-formalism mf_2 we consider both as meta-formalisms as no more capabilities are added by going to higher meta-levels.

3 Graph grammars and Meta-modelling

Graph-grammars play an important role in our approach to the modelling of complex systems. We represent models as Abstract Syntax Graphs (as a logical generalisation of Abstract Syntax Trees), and therefore model processing as graph grammars. Some of the manipulations we are interested in are:

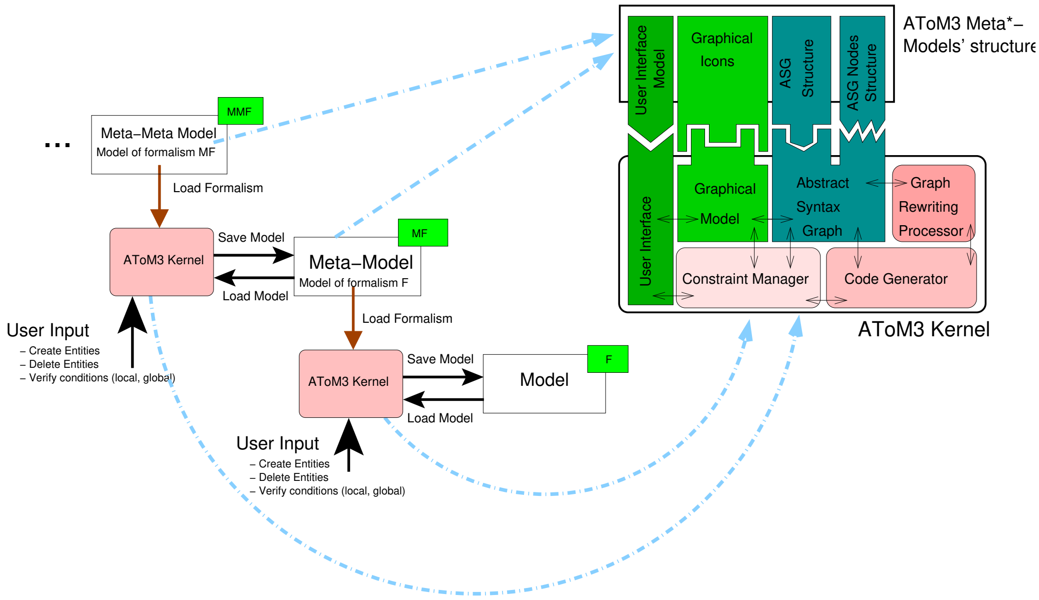


Fig. 1. Proposed working scheme for a meta-modelling environment.

- Formalism transformation: Given a model in a certain formalism, these transformations convert it into a model, but expressed in another formalism. For Modelling and Simulation, possible transformations are given in a Formalism Transformation Graph [11].
- Model optimization: These transformations do not change the formalism in which the model is expressed. Their application results in a reduction of the model complexity.
- Code Generation: These transformations produce a textual representation of the model (subject to syntactic constraints).
- Simulator specification: These graph grammars specify the operational semantics of the model. We will present an example of this kind of graph grammar in section 5.

All these tasks depend on the formalisms of interest. However, since models determined by some meta-model are graphs (subject to the constraints given by the meta-model), these tasks can be performed by a generic graph-transformation algorithm. Therefore it makes sense to combine meta-modelling and graph-grammars in a unifying framework. Meta-models determine the classes of graphs that are allowed on the LHS and RHS of a graph-grammar rule. Furthermore, the rules themselves, and the grammars, can be viewed as models in the graph-grammar formalism, which itself can be described in a meta-model.

There are tools for specifying graph-grammars and tools for meta-modelling, but to our knowledge no tool combines them in a unified framework. AToM^3 was conceived to fill this gap.

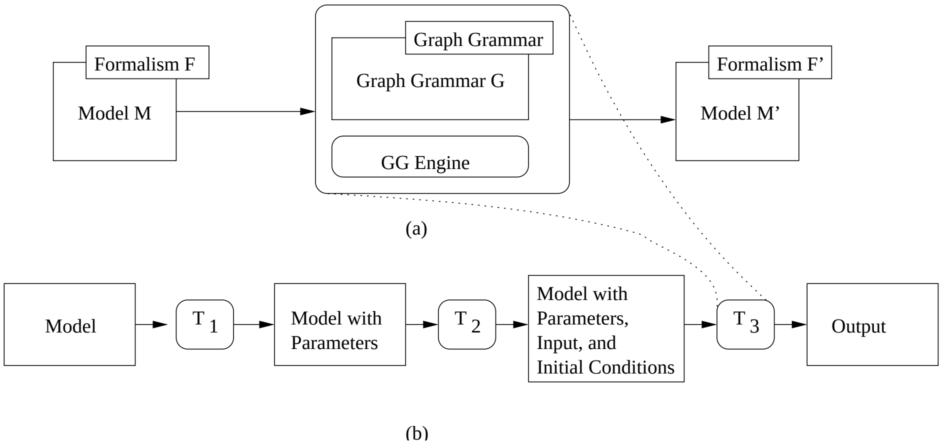


Fig. 2. Graph-grammars in Modelling and Simulation.

We would like to emphasize the role of graph-grammars in Modelling and Simulation. As mentioned before, graph transformations can be regarded as models, which can process models of other formalisms (Fig. 2 (a)). This basic paradigm can be applied to the general process of simulation. In order to simulate a model, one must first provide values to the model's parameters, and feed these, with the actual input, to the simulator ([12]). Each of these processes can be specified by graph grammars (Fig. 2 (b)). In transformations T_1 and T_2, the given model is enriched with additional structure (parameters and input). Transformation T_3 is the actual simulator, which can also be specified as a graph grammar, based on the operational semantics of the model's formalism. Input as well as output, can themselves be regarded as models in a formalism of traces (time-segments) of the values of interest.

4 AToM^3

AToM^3 is a Meta-Modelling tool written in Python [9]. Its main component is the Kernel, which is responsible for loading, saving, creating and manipulating models (at any meta-level), as well as for generating code for customized tools. Both meta-models and meta-meta-models can be loaded when AToM^3 is invoked (see Figure 1). The first kind of models allow construction of valid models in a certain formalism, the latter are used to describe the formalisms themselves. In AToM^3 all models, irrespective of meta-level, have the same internal structure (a graph).

The ERD formalism extended with constraints is available at the meta-meta-level. Constraints can be specified as OCL or Python expressions, and the designer must specify when (pre- or post- and on which event) the condition must be evaluated. Events can be *semantic* (such as editing an attribute, connecting two entities, etc.) or *graphical* (such as dragging, dropping, etc.)

When modelling at the meta-meta-level, the entities which may appear in a model must be specified together with their attributes. AToM^3 supports two kinds of attributes: *regular* and *generative*. Regular attributes are used to identify characteristics of the current entity. Generative attributes are used to generate new attributes at a lower meta-level. The generated attributes may be generative in their own right. In this way a meta-formalism, such as the ERD can be used to describe other meta-formalisms, such as the UML class diagrams. Both types of attributes, *regular* and *generative* may contain data or code for pre- and post-conditions.

The meta-meta-information is used by the *Kernel* to generate some Python files (see upper-right corner of Fig. 1), which, when loaded by the *Kernel*, allows the processing of models in the defined formalism. These files include a model of the user interface presented when the formalism is loaded. This model follows the rules of the *"Buttons"* formalism, and by default contains a button to create each object found in the meta-model. For the case of the Petri-Nets formalism ([7]), it would contain buttons to create *Places*, *Transitions*, and the connections between them. This model can be modified using AToM^3 to for example add buttons to execute graph grammars on the current model or delete unwanted buttons. When a formalism is loaded, the *Kernel* interprets this user interface model, to create and place the actual widgets and associate them with the appropriate actions.

The functionalities of the generated tools include creating models under the rules of the specified formalism, verifying that these models are valid, loading, saving, and producing a Postscript file with its graphical representation. Further model manipulations can be obtained by defining appropriate graph grammars.

For the implementation of the Graph Rewriting Processor, we have used an improvement of the algorithm given in [5], in which we allow non-connected graphs in LHSs. It is also possible to define a sequence of graph grammars that have to be applied to the model. This is useful, for example to couple grammars to convert a model into another formalism, and then apply an optimizing grammar. For clarity and efficiency reasons graph grammars are often divided in independent parts. In our tool, rules are ordered based on a user-assigned priority, and the rewriting system iteratively applies matching rules in the grammar to the graph, until no more rules are applicable.

Rule execution can either be continuous (no user interaction) or step-by-step whereby the user is prompted after each rule execution. As the LHS of a rule can match different subgraphs of the host graph, we can also control whether the rule must be applied in all the subgraphs (if disjoint), whether the user can choose one of the matching subgraphs interactively, or whetherthe system chooses a random one.

As in grammars for formalism transformations we have a mixing of entities belonging to different formalisms, it must be possible to open several meta-models at the same time. Obviously, the constraints of the individual formalism meta-models are meaningless when entities in different formalisms are present in a single model. Such a model may come to exist during the intermediate stages of graph grammar evaluation when transforming a model from one formalism into another. It is thus necessary to disable evaluation of constraints during graph grammar processing (i.e. all models are reduced to Abstract Syntax Graphs). At the end of the execution of a graph grammar for formalism transformation, the Kernel checks if the resulting model is valid in the active formalism. Formalisms used for intermediate processing are closed appropriately.

5 Meta-Modelling CBD's with AToM^3

As an example of AToM^3's capabilities to model syntax and operational semantics of formalisms, we present Causal Block Diagrams (CBD). CBD are commonly used in tools such as MathWorks' Simulink (tm).

CBDs have two basic entities: *blocks* and *links*. Blocks represent transfer functions, such as arithmetic operators or integrators. Links transmit *signals* between blocks. Signals are functions of time. We meta-model CBD syntax by means of an ERD model[1]. Our representation consists of an entity called *block* with an attribute that represents its type[2] (e.g. constant generator, or addition). Links are modelled as a relation between such entities. Links have an attribute representing the value of the signal at the current time of simulation. We also include other elements in our ERD meta-model, called *blinks*, *point*, and *focus*. They will not represent syntactic elements of the CBD per se, but structures necessary to simulate them. This is explained in more detail below. Fig. 3 shows AToM^3 with the CBD meta-model (on the left) and the generated tool to process CBD models (on the right).

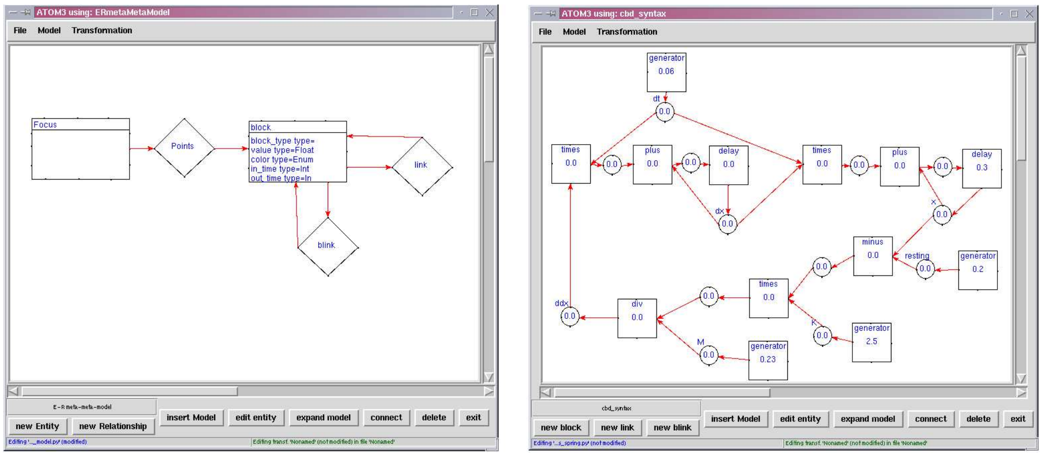


Fig. 3. ER metamodel of CBD (left), and generated tool to process CBD (right)

[1] A meta-meta-model for the ERD formalism is present in AToM^3

[2] AToM^3 has a meta-model of Types.

When simulating CBDs, unless a parallel machine is used, with a processor for each block, where all the processors work in perfect synchronization, one must choose a strategy for propagating information in a way which does not create inconsistencies. This means that there needs to be an ordering in evaluation of dependent nodes. Subgraphs that are independent could be evaluated concurrently, but only before any block that is influenced by them.

The solution is simple: 1) order the nodes by a topological sort of the graph (done by a standard depth-first traversal) 2) evaluate each block following this ordering. In section 6.1 we present such an algorithm by means of a graph grammar. For this we require an additional type of link between blocks, which we call *blink*. A blink between a block B_1 and a block B_2 represents the relation "evaluate B_1 before evaluating B_2". After the topological sort has finished, there will be a hamiltonian path over the blocks where the blocks will be connected by edges of type *blink*. The other entity, the *focus* is a pointer to the block being processed. Only one focus entity is created by the graph grammar, since our approach is purely sequential.

5.1 CBD Denotational Semantics

Here, we provide an informal description of the denotational semantics of block diagrams (Figure 4 on the left). This description simply associates each block diagram to a set of equations representing the values of the links between blocks as signals. More precisely, the denotation of a block diagram is the set of signal functions corresponding to every link in the diagram [3].

In order to simulate CBD on a digital computer we need to discretize the signals, i.e. use the natural numbers as the time-base for the signals. The interpretation of the delay block adopted here is only for a discrete time-base. The other blocks have the same interpretation for both discrete and continuous time. The denotation for the delay block in continuous time has to take a time segment as initial condition, instead of the point value h, shown here.

6 Processing CBD's models

By having the natural numbers as our time base, we can view signals as streams, i.e. unbounded sequences of the values that the signals take at the discrete points. This allows us to see the block diagram as a dataflow network. Here we present an approach based on [1]. One way to model this is by providing each link with an attribute that represents the complete stream computed so far. By doing so, the definition of the operational semantics of CBDs by means of a graph grammar becomes straight-forward. Certaintly this is space-expensive but, as we mentioned in the introduction, the goal of this approach is not to achieve

[3] In our treatment of causal block diagrams we require the explicit use of delay blocks whenever there is a feedback loop. The reason is that otherwise, the denotational semantics given here would produce inconsistencies in the presence of such loops. Furthermore we require there to be at least one constant generator in the model.

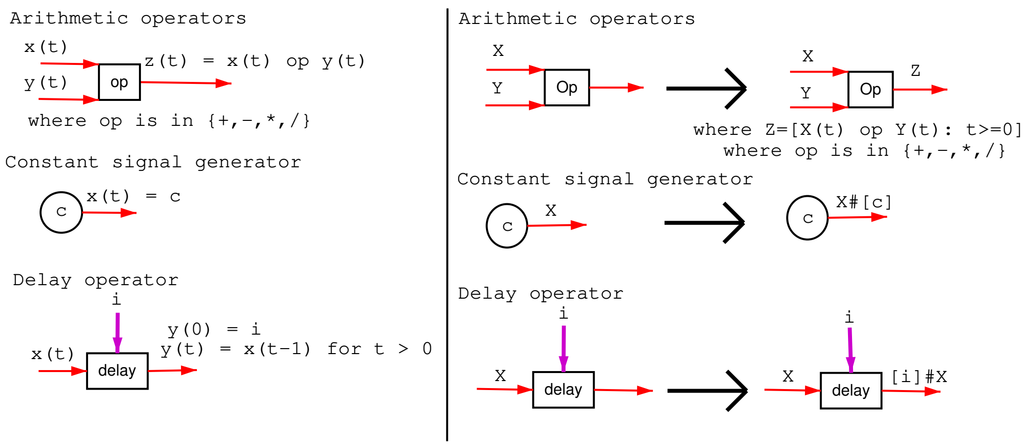


Fig. 4. Denotational (left) and Operational (right) semantics of CBD's.

efficiency, but to be able to define an *executable specification* of the operational semantics.

A first, stream-oriented approach to the operational semantics is straightforward. We observe the following conventions: uppercase letters represent streams, explicit streams are written as lists with square brackets, e.g. $[x_0, x_1, x_2, ...]$. Stream concatenation is done with the $\#$ operator. If we have a finite stream X, then $X\#[e]$ represents the stream resulting from appending e to X. The operational semantics are defined then as shown in figure 4 (right).

This matches the denotational semantics: The constant generator simply generates an infinitely long stream: $X = [c, c, c, ...]$. Hence $X(t) = c$ for all $t \geq 0$. The rule for an arithmetic operator block $*$ guarantee that if $X = [x_0, x_1, x_2, ...]$ and $Y = [y_0, y_1, y_2, ...]$ then $Z = [x_0 * y_0, x_1 * y_1, x_2 * y_2, ...]$, that is, $Z(t) = X(t) * Y(t)$ for all $t \geq 0$. Finally, for the delay operator we have that if the input is $X = [x_0, x_1, x_2, ...]$ then the output is $Y = [i, x_0, x_1, x_2, ...]$, i.e. $Y(0) = i$, $Y(1) = X(0)$, $Y(2) = X(1)$, etc. Hence $Y(t) = X(t-1)$ for all $t > 0$.

6.1 Topological Sort

The problem with these rules is that they do not take into account the issue of evaluation order. This might be enough to reason about CBDs, but not to produce an "executable specification". In order to deal with this, we introduce a set of rules which will sort the blocks. This set of rules is to be evaluated before the actual operational semantics rules.

The general idea of this set of rules is based on a depth-first search of the graph [3]. Here we explicitly construct the evaluation path, i.e. we create blinks between the blocks. First we find some root block[4], a block without parents, and

[4] In the CBD presented here there will always be at least one root node, since we require to be at least one constant generator.

visit all its children recursively, marking with a colour blocks already visited. When a node has not been visited, it is white. When it has been visited but not all its decendants have been explored, it is gray. Otherwise it is black. We also keep a pointer to the node currently visited, which we call the *focus*. As the focus goes from parent to child, a *blink* edge is created between them, and a blink coming out of the parent is transferred to the child. When backtracking after finding a dead-end (i.e. a gray or black node, already visited), blinks are left unchanged. When a branch has been completely explored, a new root is searched for and the process is repeated until all nodes are coloured black.

Given the space limitations only some representative rules are shown (for details, we refer to the AToM^3 web-page [2]). Blocks will have a counter representing the number of immediate children being explored. The rule in Fig. 5 shows the rule representing the discovery of a node that hasn't been visited, as described above. (Dashed arrows are blinks.)

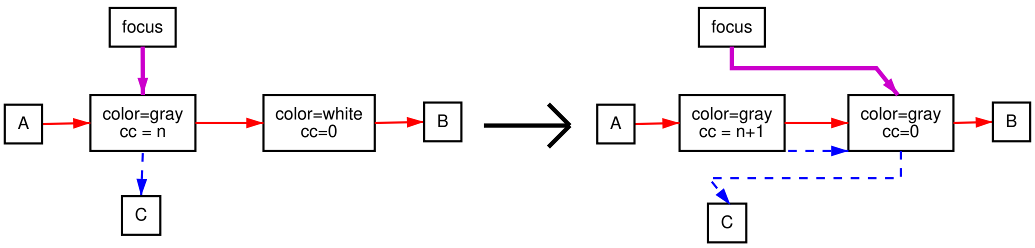


Fig. 5. Topological sort of a CBD: new non-terminal discovered

Another important rule is shown in Fig. 6, depicting the backtracking when a loop is detected or when all the children have been visited.

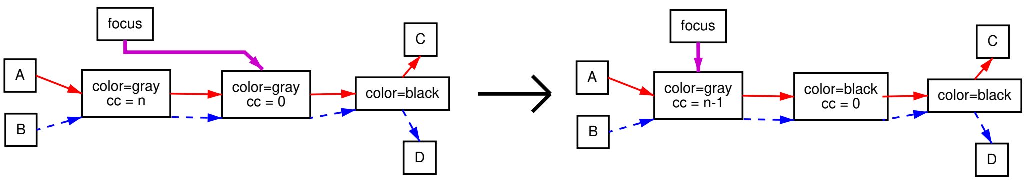


Fig. 6. Topological sort of a CBD: loop detected or children completed.

The graph-grammar implementing the topological sort, adds a blink between the last node and the first, making it a loop.

6.2 Operational Semantics

We need to adapt the rules shown in Fig. 4 so that blocks are evaluated in the correct order. This is implemented by focusing on one block, evaluating it, and follow the blinks created by the topological sort. An example of one such rule is shown in figure 7.

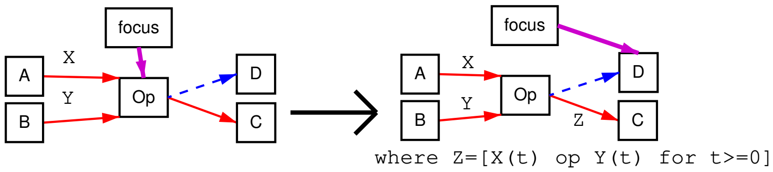


Fig. 7. A representative rule for evaluation of a CBD.

Since the topological sort returns a loop covering all nodes, evaluation proceeds following the described scheme until some termination criteria is met. To specify termination, we add two global attributes to the meta-model of CBD: an iteration counter, and a maximum number of iterations attribute. Models in AToM^3 can also have user-defined constraints, making it easy to define the termination criteria in terms of these atributes.

6.3 Simulation Results

The specified CBD simulator was tested on the harmonic oscillator equation (also known as the "circle test"): $\frac{dx^2}{dt^2} = -x, x(0) = 1, \frac{dx}{dt}(0) = 0$. Full results can be found on the $AToM^3$ homepage [2].

7 Conclusions

In this article we have presented our approach to modelling complex systems, which is based on meta-modelling and multi-formalism modelling, and is implemented in the software tool AToM^3. This code-generating tool, developed in Python, relies on graph grammars and meta-modelling techniques.

We have demonstrated how both syntax and operational semantics of the commonly used formalism Causal Block Diagrams formalism can be modelled. When doing so in AToM^3, a tool for modelling and simulating CBD is automatically obtained.

Our main contribution is the unification of meta-modelling (formalisms – classes of models – may be modelled in their own right) and graph transformation based on graph grammar specifications.

The advantages of using an automated tool for generating customized model-processing tools are clear: instead of building the whole application from scratch, it is only necessary to specify –in a graphical manner– the kinds of models we will deal with. The processing of such models can be expressed at the meta-level by means of graph grammars.

AToM^3, with meta-models for modelling with Entity-Relationship, Data Flow Diagrams, Structure Charts, Petri-Nets, Statecharts, GPSS, DEVS and Finite State Automata and some transformations is available at [2].

Acknowledgements

This paper has been partially sponsored by the Spanish Interdepartmental Commission of Science and Technology (CICYT), project number TEL1999-0181. Prof. Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

References

1. Abelson H., Sussman G. J. *Structure and Interpretation of Computer Programs* 2nd edition. MIT Press. 1996.
2. AToM^3 Home page: http://moncs.cs.mcgill.ca/MSDL/research/projects/ATOM3.html
3. Cormen, T., Leiersson, C. H., Rivest, R. S. *Introduction to Algorithms* 1st edition. MIT Press. 1990.
4. DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center. Honeywell, 1999, version 5.2.1
5. Dorr, H. 1995. *Efficient Graph Rewriting and its implementation.* Lecture Notes in Computer Science, 922. Springer.
6. Gray J., Bapty T., Neema S. 2000. *Aspectifying Constraints in Model-Integrated Computing*, OOPSLA 2000: Workshop on Advanced Separation of Concerns, Minneapolis, MN, October, 2000.
7. Murata, T. *Petri Nets: Properties, analysis and applications.* Proceedings of the IEEE, 77(4)-541-580.
8. OMG Home Page: *http://www.omg.org*
9. Python home page: *http://www.python.org*
10. Sztipanovits, J., et al. 1995. *"MULTIGRAPH: An architecture for model-integrated computing"*. In ICECCS'95, pp. 361-368, Ft. Lauderdale, Florida, Nov. 1995.
11. Vangheluwe, H. *DEVS as a common denominator for multi-formalism hybrid systems modelling.* In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134. IEEE Computer Society Press, September 2000. Anchorage, Alaska.
12. Zeigler, B., et al. 2000 *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press. Second Edition. 2000

Application of Graph Transformation Techniques to the Area of Petri Nets: An Overview ⋆

B. Braatz, H. Ehrig, K. Hoffmann, J. Padberg, M. Urbášek

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

1 Aims and Introduction

The main aim of this contribution is to give an overview concerning applications of graph transformation techniques to the area of Petri nets achieved by the team of TU Berlin within the APPLIGRAPH Working Group and the DFG Researcher Group on Petri Net Technology.

Since about 10 years the strong relationships between the areas of Petri nets and graph transformation systems have been studied especially as part of the cooperation of the groups in Pisa and Berlin, while it was observed by Kreowski already in the early 80s how Petri nets can be considered as a special case of graph transformation systems. The main aim of the Pisa-Berlin cooperation was to transfer the well-known constructions leading to a truly concurrent event structure semantics from Petri nets to graph transformation systems. Vice versa the concept of high-level replacement systems, short HLR-systems, by Ehrig, Habel, Kreowski and Parisi-Presicce in [EHKP91] was the starting point to obtain new concepts and results for the area of Petri nets by application of graph transformation techniques. In fact the concept of HLR-systems in [EHKP91] is a generalization of the double pushout approach from graph transformation to HLR-systems in a categorical framework.

The instantiation of HLR-systems to Petri nets leads to the concept of net transformation systems [PER95], which will be discussed in Section 2. In order to study property preserving transformations, the concept of HLR-transformations was extended by Padberg in [Pad96] to $\mathcal{Q}$-transformations leading to safety and liveness preserving transformations in [PGE98,GHP99] and [GPU01] respectively reviewed in Section 3. In Section 4 we discuss in which way the module concept for graph transformation systems developed by Simeoni [Sim00] has been transferred to Petri nets [PHBS02] and to a generic component concept in [EOBKP02]. In Section 5 we discuss the modeling of open systems, where the concept of open graph transformation systems developed by Heckel [Hec98] has

⋆ This work is partially supported by the project APPLIGRAPH (ESPRIT Basic Research WG), GRAPHIT (CNPq and DLR) and by the joint research project "DFG-Forschergruppe PETRINETZ-TECHNOLOGIE" between H. Weber (Coordinator), H. Ehrig (both from the Technical University Berlin) and W. Reisig (Humboldt University Berlin), supported by the German Research Council (DFG).

influenced the development of open nets in [BCEH01]. Finally in Section 6 we briefly mention other topics, where the areas of graph transformation techniques and Petri nets have influenced each other.

2 Net Transformation Systems as Instantiation of High-Level Replacement Systems

The general idea of high-level replacement (HLR) systems is to generalize the concept of graph transformation systems and graph grammars from graphs to all kinds of structures which are of interest [EHKP91]. This generalization has been done categorically and can be applied to all kinds of high-level structures, especially also different kinds of Petri nets. Several results from graph grammars have been reformulated in the framework of high-level replacement systems and can be applied to other high-level structures without the necessity to be proven again. The theory of HLR systems is based on the double pushout approach, which has been widely investigated in the area of graph grammars (see [Ehr79]). The HLR framework is suitable for many high-level structures. The concept of transformations has been applied to several classes of Petri nets (P/T Petri nets, colored Petri nets, AHL nets), yielding the idea of net transformation systems first introduced in [PER95].

The next definition introduces rules, transformations and net transformation systems formally for a given category **NET** of low or high level nets. More about the underlying theory can be found in [PER95] and in [Pad99].

Definition (Rules, Transformations and Transformation Systems).

1. *A* rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *in a category* **NET** *consists of the objects* L*,* K *and* R*, called left-hand side, interface (or gluing object), and right-hand side, respectively, and two morphisms* $K \xrightarrow{l} L$ *and* $K \xrightarrow{r} R$ *with both morphisms* $l, r \in \mathcal{M}$*, a suitable class of injective morphisms in* **NET**.
2. *Given a rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$*, a* direct transformation $G \overset{p}{\Longrightarrow} H$ *from a net* G *to a net* H *is given by two pushout diagrams* **(1)** *and* **(2)** *in the category* **NET** *as shown below.*

$$\begin{tikzcd} L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\ G & C \arrow[l, "g"] \arrow[r, "h"'] & H \end{tikzcd}$$

   The morphisms $L \xrightarrow{m} G$ *and* $R \xrightarrow{n} H$ *are called occurrences of* L *in* G *and* R *in* H*, respectively. By an occurrence of rule* $r = (L \xleftarrow{l} K \xrightarrow{r} R)$ *in a net* G *we mean an occurrence of the left-hand side* L *in* G*.*
   In fact, the occurrence morphism m *has to satisfy a specific condition, called gluing condition, in order to apply the rule* p *to the net* G*.*
3. *Given a category of nets* **NET** *together with a suitable class of injective morphisms* $\mathcal{M}$*, a* net transformation system $H = (\mathcal{S}, \mathcal{P})$ *in* $(\mathbf{NET}, \mathcal{M})$ *is given by a start net* $\mathcal{S} \in |\mathbf{NET}|$*, and a set of rules* $\mathcal{P}$*.*

The idea of net transformation systems is the basic idea behind the stepwise development of communication based systems in the framework of Petri nets. Each transformation step can be formally depicted as a rule-based transformation according to an appropriate rule in a specific net transformation system. The transformation sequence then provides a transformation from the initial net G to the final net H as shown below.

$$G = G_0 \overset{p_1}{\Longrightarrow} G_1 \overset{p_2}{\Longrightarrow} \ldots \overset{p_n}{\Longrightarrow} G_n = H.$$

Several results concerning horizontal structuring for net transformation systems have been adopted from HLR systems. The two basic structuring constructions for nets are union and fusion. Union allows a construction of larger nets from smaller ones with shared subpart, while fusion is a construction which allows to identify distinguished subnets.

Especially the concept of fusion is more general then the concept of fusion of places often cited in the Petri net literature. The fusion introduced in net transformation systems is not restricted to the fusion of places, but covers also fusion of subnets.

In a view of software development methodology it is important that horizontal structuring based on fusion and union is compatible with transformations. This means that – under certain compatibility conditions – the result is the same whether we apply first union/fusion and then transformation or vice versa. For more details about horizontal structuring and transformations see e.g. [Pad96,Pad99].

This net transformation technique including horizontal structuring has been successfully applied in the case study of a larger medical information system summarized in [EPE96].

3 Property Preserving Net Transformations

Although the net transformation framework is a suitable concept for stepwise development of systems, very often there is a need to consider in addition more general morphisms for refinement or abstraction. The main idea is to enlarge the category of nets by $\mathcal{Q}$-morphisms in the sense of [Pad96] in order to formulate refinement/abstraction morphisms.

More precisely, another category of nets **QNET** with a distinguished class of morphisms $\mathcal{Q}$, called $\mathcal{Q}$-morphisms, is employed. The category **NET** of nets from the previous definition in Section 2 has to be a subcategory of **QNET**. The class of $\mathcal{Q}$-morphisms is the class of refinement/abstraction morphisms. This class of morphisms has to satisfy additional requirements called $\mathcal{Q}$-conditions (see [Pad96]) to be adequate for refinement or abstraction.

Then, a single transformation step is formally given by Fig. 1. The squares **(1)** and **(2)** represent a transformation in the category **NET** (as in definition in Section 2). A morphism q is the refinement/abstraction morphism in **QNET**, such that $q \in \mathcal{Q}$. When the $\mathcal{Q}$-conditions are satisfied then there exists an induced

$$
\begin{tikzcd}
L \arrow[d, "m"'] \arrow[rr, bend left, "q"] \arrow[dr, phantom, "(\mathbf{1})"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(\mathbf{2})"] & R \arrow[d, "n"] \\
G \arrow[rr, bend right, "q'"'] & C \arrow[l, "g"] \arrow[r, "h"'] & H
\end{tikzcd}
$$

Fig. 1. Transformation step

$\mathcal{Q}$-morphism $q' \in \mathcal{Q}$ in **QNET**, which is a morphism between the original and the transformed net.

The idea of $\mathcal{Q}$-morphisms for net transformation systems has also been introduced for HLR systems in general. It is a powerful formal technique for design of complex systems. However, it has been applied up to now only in the domain of Petri nets. The application to graph transformation systems is certainly also of interest and a point of future investigation.

The concept of $\mathcal{Q}$-morphisms is important in order to study whether the transformation of nets is property preserving. During the transformation process, a net may become too large to check some properties of this net efficiently. If this property could be checked or stated for an initial net before the transformation starts and then preserved during the transformation process, a tedious investigation of properties for the final net can be omitted.

The idea of property preserving transformations has been investigated in [GHP99,PG00,PGE98] for safety properties and in [GPU01] for liveness. Safety-properties for Petri nets are stated as propositional logic formulas upon the actual marking of Petri nets. Morphisms preserving safety properties have been investigated for several types of Petri nets, see [GHP99] for P/T Petri nets, [PG00] for colored Petri nets and [PGE98] for a class of algebraic-high level nets. This class of morphisms has been applied also in a case study of a medical information system in [Pad99] in order to prove relevant properties of the information system.

Liveness preserving refinement is based on the standard notion of liveness as used in Petri net theory. Liveness means that no deadlock or even livelock can occure. In [GPU01] it is shown that a special type of transition refinement preserves liveness in Petri nets. The idea is based on abstracting morphisms, which are related to vicinity respecting morphisms (introduced in [DM90]). A certain subclass of abstracting morphisms, called a class of collapsing morphisms, allows a description of a transition refinement as collapsing of a subnet to one transition. The preservation of liveness has been proven for collapsing morphisms and demonstrated on an example in [GPU01].

Both types of property preserving transformations give an oportunity to cut the cost of verification of system properties for large systems.

4 Module and Component Concepts

A variety of modularity concepts has been introduced for graph transformation systems in the literature. An overview and classification is given in [HEET99]. In the following we consider the concept of M. Simeoni introduced in [Sim00] which has been adapted also to Petri nets.

Modules of graph transformation systems in the sense of [Sim00] are based on refinement morphisms *exp* between export EXP and body BOD of a module and injections *imp* between the import IMP and the body (cf. Fig. 2). It is important to notice that IMP, EXP and BOD are graph transformation systems rather than single graphs. The refinement morphism represents the elaboration of certain aspects of the graph transformation system in the export interface to greater detail in the body. The import interface contains the parts of the body to be further refined.

$$\begin{tikzcd} & EXP \arrow[d, "exp"] \\ IMP \arrow[r, hook, "imp"] & BOD \end{tikzcd}$$

Fig. 2. Module with *exp*-refinement and *imp*-inclusion

It is shown in [Sim00] that the category consisting of typed algebraic graph transformation systems as objects and refinements as morphisms has pushouts if at least one of the morphisms is an inclusion. This means that refinements of the import of a module yield corresponding refinements of the body.

In [PHBS02] this approach is applied to Petri nets leading to Petri net modules. In this case IMP, EXP and BOD are Petri nets. Refinement morphisms are defined as a more general kind of Petri net morphisms allowing transitions not only to be mapped to transitions, but also to whole subnets of the target net. As in the case of graph transformation systems composition of modules can be defined by lifting of a refinement from the import to the body.

Abstracting the ideas in [Sim00] and [PHBS02] a concept of components for generic modeling techniques is introduced in [EOBKP02], which allows arbitrary kinds of transformations between the export and the body of a component. The only property that must be satisfied by the chosen transformation concept is the extension property, which requires that transformations can be lifted along an inclusion.

Moreover, a transformation semantics is proposed which is suitable for semiformal modeling techniques like the UML, as well as for formal modeling techniques with tight semantics like Petri nets. This semantics is a function, yielding for every transformation of the import of a component a corresponding transformation of the export. The transformation semantics is shown to be compositional, i. e. the semantics of a composed component can be obtained from the semantics of its parts. These results of [EOBKP02] can be transferred back to Petri nets as well as graph transformation systems.

5 Modeling of Open Systems

In the double-pushout (DPO) approach, a graph derivation $G \overset{p,m}{\Longrightarrow} H$ is uniquely determined up to isomorphism for a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ by an occurence morphism $m\colon L \to G$ and the requirement that **(1)** and **(2)** in Fig. 3 are pushout diagrams. Similar to other graph transformation systems this can be considered as a tight semantics assigned to rules.

$$
\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "\mathbf{(1)}"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "\mathbf{(2)}"] & R \arrow[d, "n"] \\
G & C \arrow[l, "g"] \arrow[r, "h"'] & H
\end{tikzcd}
$$

Fig. 3. Double pushout resp. double pullback diagram

The modeling of open systems, however, requires a kind of loose semantics, because in addition to the changes specified by the rules, unspecified changes can occur due to interaction of the system with the environment or a user. In [Hec98] this loose semantics is achieved by a double-pullback (DPB) approach leading to graph transitions $G \overset{p/d}{\longrightarrow} H$. In contrast to the DPO approach, the requirement that **(1)** and **(2)** in Fig. 3 are pullback diagrams leaves changes to the environment of the occurence unspecified. Therefore a graph transition is not uniquely defined by a morphism $m\colon L \to G$ but only by a triple of morphisms $d = \langle d_L, d_K, d_R \rangle$, determining the implicit changes to the environment.

A second important concept of [Hec98] is the explicit frame condition, restricting the parts of the graph on which interaction with the environment can take place to output and input sorts O^-, O^+ and requiring transitions to be direct derivations for the remaining parts. This approach is applied to Petri nets by the notion of open Petri nets [PJHE98]. An open net consists of an ordinary Petri net N and sets of output and input places O^-, O^+, on which tokens can be exchanged with the environment. This concept has been formally introduced in [BCEH01] where open nets are used to model two workflows interacting with each other by a common interface. The open places in either of the nets for the two workflows represent the environment through which communication with the other one is possible.

As a main result in [BCEH01] a Goltz-Reisig process semantics is introduced for open nets. For these processes an amalgamation much like the amalgamation of algebras (see e. g. [EM85]) is developed, which can be used to achieve compositionality of the semantics of open nets. It seems promising to transfer these results back to graph transformation systems. This, however, is future work.

The concept of inclompete information in open systems leads to the framework presented in [EHLOPR00] for generic rule-based modeling techniques, which is instantiated on one hand by the graph transitions mentioned above, on the other hand by open Petri nets, where rules correspond to net transitions $t\colon pre(t) \Rightarrow post(t)$, transformations to the firing of a net transition and transitions to the firing of a net transition with unspecified output and input on the open places.

6 Further Recent Developments

In [Hof00,Hof01] we have introduced the concept and formal definition of Algebraic Higher Order Nets where the data type part is extended by higher order types, sorts and functions. This allows to have functions as data items on places, such that different functions may be activated and applied during run time. As far as we can see it is possible to extend analogously the data type part of attributed graph transformation leading to higher order attributed graph transformation. As part of ongoing work we will describe on one hand the theory of higher order attributed graph transformation and on the other hand the application domain where this feature of flexible modeling is especially useful.

A further recent development based on transformations for arbitrary specification techniques in the sense of high-level replacement systems is the description of architecture evolution. The architecture of specifications is represented by a graph that is used as a diagram over the specification. These diagram functors can then be transformed in the usual double-pushout approach. The formal foundation is given in [Pad01]. An informal version where an extensive example and its implementation in GenGEd is discussed can be found in [BEQ02].

7 Conclusion

In this paper we have presented four main topics where techniques from the area of graph transformation systems have influenced new developments in the area of Petri nets. The most promising ones for further development are the property preserving transformations in Section 3 and the module and component concepts in Section 4, because they support stepwise construction and verification of large systems from small components in the framework of Petri nets. An additional topic discussed in another overview paper [BEEQW02] are visual modeling techniques based on attributed graph transformation systems [1] which have been applied to Petri nets leading to an interesting new concept of animation of Petri nets [7, BEEQW02].

References

[Bar99] R. Bardohl. *Visual Definition of Visual Languages based on Algebraic Graph Transformation.* PhD thesis, Technische Universität Berlin, 1999. Published by Verlag Dr. Kovac, 2000.

[BCEH01] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional Modeling of Reactive Systems Using Open Nets. In *Proc. CONCUR 2001*, Springer LNCS 2154, pages 502–518. Springer Verlag, 2001.

[BEEQW02] R. Bardohl, K. Ehrig, C. Ermel, A. Qemali, and I. Weinhold. Specifying Visual Languages with GenGEd. In *Proc. AGT 2002: APPLIGRAPH Workshop on Applied Graph Transformation*. 2002.

[BEQ02] R. Bardohl, C. Ermel, and A. Qemali. Transforming Specification Architectures with GenGEd. Submitted, 2002.

[DM90] J. Desel and A. Merceron. Vicinity Respecting Net Morphisms. In *Advances in Petri Nets*, Springer LNCS 483, pages 165–185. Springer Verlag, 1990.

[EBE01] C. Ermel, R. Bardohl, and H. Ehrig. Specification and Implementation of Animation Views for Petri Nets. In *Proc. 2nd Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, pages 75–92. 2001.

[EHKP91] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From Graph Grammars to high level replacement systems. In *Proc. 4th Int. Workshop on Graph Grammars and their Application to Computer Science*, Springer LNCS 532, pages 269–291. Springer Verlag, 1991.

[EHLOPR00] H. Ehrig, R. Heckel, M. Llabrés, F. Orejas, J. Padberg, and G. Rozenberg. Double-Pullback Graph Transitions: A Rule-Based Framework with Incomplete Information. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation*, Springer LNCS 1764, pages 85–102. Springer Verlag, 2000.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Proc. 1st Graph Grammar Workshop*, Springer LNCS 73, pages 1–69. Springer Verlag, 1979.

[EM85] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Vol. 6 of *EATCS Monographs on Theor. Comp. Science*. Springer Verlag, 1985.

[EOBKP02] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A Generic Component Concept for System Modeling. In *Proc. FASE '02*. 2002.

[EPE96] C. Ermel, J. Padberg, and H. Ehrig. Requirements Engineering of a Medical Information System Using Rule-Based Refinement of Petri Nets. In *Proc. IDPT '96*, pages 186–193. 1996.

[GHP99] M. Gajewsky, K. Hoffmann, and J. Padberg. Place Preserving and Transition Gluing Morphisms in Rule-Based Refinement of Place/Transition Systems. Technical Report 1999-14. Technical University Berlin, 1999.

[GPU01] M. Gajewsky, J. Padberg, and M. Urbášek. Rule-Based Refinement for Place/Transition Systems: Preserving Liveness-Properties. Technical Report 2001-8. Technical University Berlin, 2001.

[Hec98] R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems*. PhD thesis, Technische Universität Berlin, 1998.

[HEET99] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. Classification and Comparison of Modularity Concepts for Graph Transformation Systems. In *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.

[Hof00] K. Hoffmann. Run Time Modification of Algebraic High Level Nets and Algebraic Higher Order Nets using Folding and Unfolding Construction.

In *Proc. 3rd Int. Workshop Communication Based Systems*, pages 55–72. 2000.

[Hof01] K. Hoffmann. Flexible Modellierung mit Algebraischen Higher Order Netzen. In *Proc. Workshop Modellierung*, pages 101–110. 2001.

[Pad96] J. Padberg. *Abstract Petri Nets: A Uniform Approach and Rule-Based Refinement*. PhD thesis, Technische Universität Berlin, 1996. Published by Shaker Verlag.

[Pad99] J. Padberg. Categorical Approach to Horizontal Structuring and Refinement of High-Level Replacement Systems. In *Applied Categorical Structures*, 7(4):371–403. 1999.

[Pad01] J. Padberg. *Formal Foundation for Transformations of Specification Architectures*. Technical Report. Technical University Berlin, 2001.

[PER95] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic High-Level Net Transformation Systems. In *Mathematical Structures in Computer Science*, 2:217–256. 1995.

[PG00] J. Padberg and M. Gajewsky. Safety Preserving Transformations of Coloured Petri Nets. Technical Report 2000-13. Technical University Berlin, 2000.

[PGE98] J. Padberg, M. Gajewsky, and C. Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. In *Proc. FASE '98*, Springer LNCS 1382, pages 221–238. Springer Verlag, 1998.

[PHBS02] J. Padberg, K. Hoffmann, M. Buder, and A. Sünbül. Petri Net Modules for Component-Based Software Engineering. Technical Report. Technical University Berlin, 2002.

[PJHE98] J. Padberg, L. Jansen, R. Heckel, and H. Ehrig. Interoperability in Train Control Systems: Specification of Scenarios using Open Nets. In *Proc. IDPT '98*, pages 17–28. Society for Design and Process Science, 1998.

[Sim00] M. Simeoni. *A Categorical Approach to Modularization of Graph Transformation Systems using Refinements*. PhD thesis, Università di Roma La Sapienza, 2000.

Proving Distributed Algorithms by Graph Relabelling Systems: Examples of Trees in Networks with Processor Identities *

Y. Métivier, M. Mosbah, and A. Sellami

LaBRI
Université Bordeaux1 - ENSEIRB
351 Cours de la Libération
33405 - Talence, France.
{metivier, mosbah, sellami}@labri.fr

1 Introduction

Graph Relabelling Systems (GRS) have been introduced in [4] as a suitable tool for encoding distributed algorithms, for proving their correctness and for understanding their power. In this model, a network is represented by a graph which vertices denote processors, and edges denote communication links. The local state of a processor (resp. link) is encoded by the label attached to the corresponding vertex (resp. edge). A relabelling rule is a rewriting rule which has the same underlying fixed graph for its left-hand side and its right-hand side, but with an update of the labels. According to its own state and to the states of its neighbours, each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbours and of the corresponding edges may have changed according to some specific *computation rules*. Thus they satisfy the following requirements:

(C1) they do not change the underlying graph but only the labelling of its components (edges and/or vertices), the final labelling being the result,
(C2) they are local, that is, each relabelling changes only a connected subgraph of a fixed size in the underlying graph,
(C3) they are locally generated, that is, the applicability condition of the relabelling only depends on the local context of the relabelled subgraph.

For such systems, the distributed aspect comes from the fact that several relabelling steps can be performed simultaneously on "far enough" subgraphs, giving the same result as a sequential realization of them, in any order. A large family of classical distributed algorithms encoded by graph relabelling systems is given in [2, 3].

The power of relabelling systems can be increased using priorities (PGRS) or forbidden contexts (FCGRS). A priority relation (see [4, 8, 9]) is a partial

* This work has been supported by the TMR research network GETGRATS and by the "Conseil Régional d'Aquitaine"

order on the set of relabelling rules. In PGRS systems, the priority relation determines the order in which rules have to be applied if they overlap. In FCGRS [10], relabelling rules are equipped with forbidden contexts, that is graphs which disable the application of the corresponding rule if they are included in the context.

In this paper, we deal with relabelling systems for named networks. Processors are assumed to have distinct identities. We show that relabelling systems can be applied for those networks. We consider the examples of computing spanning trees in such networks. By applying various types of relabelling systems, we capture various paradigms of the distributed computation of a spanning tree. These paradigms include sequential or distributed computation, local detection of the global termination, and correctness of computation.

One fundamental problem in distributed computing is the termination detection property [14]. That is, the existence of a vertex which decides whether the algorithm is globally terminated. In fact, it is not trivial for a processor to have an up to date knowledge on the global state of the system, and to decide whether a distributed computation has finished. There are two types of termination. In an implicitly terminating algorithm, each execution is finite and in the last state of the execution each node has the correct result. However, the nodes are not aware of the global termination. Termination is said to be explicit in a process if that process is in a terminal configuration and its state indicate the termination of the algorithm. There were many proposals for *termination detection* algorithms: such algorithms transform implicitly into explicitly terminating algorithms. Several conditions were found to allow such algorithms and for each of these conditions a specific algorithm was given (see [14]). In this paper, by termination we mean explicit termination.

On one hand, we will give examples of relabelling systems which have the termination detection property, on the other hand we will give a method which transforms a relabelling system which is initialized by exactly one vertex into another one with the termination detection property. For the latter, we will use the well-known Dijkstra-Scholten algorithm [6] to detect termination. Such an algorithm fits in our framework, and can be encoded by a graph relabelling system. Moreover, by combining the relabelling system encoding the Dijkstra-Scholten algorithm with another representing a given distributed algorithm, we obtain a graph relabelling system encoding the latter algorithm which has the termination detection property. We illustrate this method by an example which computes a spanning tree.

We base our examples on the computation of a spanning tree in a network, because it is among the important problems in distributed computing. Trees are essential structures in various communication protocols (synchronization, deadlock resolution, information broadcasting). This problem is also closely related to the election problem. The aim of an election in a network of processors is to choose exactly one element in the set of processors: this element is called elected or leader. The leader can be used subsequently to make decisions or to centralize some information. In fact, the problem of finding a leader is reducible to the problem of finding a spanning tree. For anonymous networks, computing spanning trees by graph rewriting techniques has already been investigated in [8]. We focus in this paper on networks with processor identities. Let us note that all algorithms presented in this paper have been implemented and tested [3, 12, 2].

Among models related to our model there are local computation systems as defined by Rosenstiehl and al. [13], Angluin [1], Yamashita and Kameda [7] and Boldi and Vigna [5]. In [13] a synchronous model is considered, where vertices represent (identical) deterministic finite automata. The basic computation step is to compute the next state of each processor according to its state and the states of its neighbours. In [1] an asynchronous model is considered. A basic computation step means that two adjacent vertices exchange their labels and then compute new ones. In [7] an asynchronous model is studied where a basic computation step means that a processor either changes its state and sends a message or it receives a message. In [5], networks are directed graphs colored on their arcs; each processor changes its state depending on its previous state and on the states of its in-neighbours. Activation of processors may be synchronous, asynchronous or interleaved.

The paper is organized as follows. Section 2 presents relabelling systems by giving examples of computing spanning trees in a network with processor identities. In Section 3, we show how to add termination detection property to a relabelling system which is initialized by exactly one vertex. A detailed version of this paper with proofs and other examples can be found in [11].

2 Computation of a Spanning Tree in a Network with Processor Identities

2.1 Techniques for Proving Distributed Computing

Graph relabelling systems provide a formal model for expressing distributed algorithms. The aim of this section is to show that this model is suitable for studying and proving properties of distributed algorithms.

An *L-labelled graph* is a graph whose vertices and edges are labelled with labels from a possibly infinite alphabet L. It will be denoted by $G(V, E, \lambda)$, where G is a graph and $\lambda : V(G) \cup E(G) \to L$ is the *labelling function.* The graph G is called the *underlying graph* of $G(V, E, \lambda)$, and λ is a *labelling* of G. The class of L-labelled graphs will be denoted by $\mathcal{G}_L$. A graph relabelling system $\mathcal{R}$ is *noetherian* if there is no infinite $\mathcal{R}$-relabelling sequence starting from a graph with initial labels in $\mathcal{I}$. Thus, if a distributed algorithm is encoded by a noetherian graph relabelling system then this algorithm always terminates. In order to prove that a given system is noetherian we generally use the following technique. Let $(S, <)$ be a partially ordered set with no infinite decreasing chain (that is every decreasing chain $x_1 > x_2 > \cdots > x_n > \dots$ in S is finite), we say that $<$ is a *noetherian order*. It is compatible with $\mathcal{R}$ if there exists a mapping f from $\mathcal{G}_L$ to S such that for every $\mathcal{R}$-relabelling step $(G, \lambda) \to (G, \lambda')$ we have

$f(G,\lambda) > f(G,\lambda')$. It is not difficult to see that if such an order exists then the system $\mathcal{R}$ is noetherian: since there is no infinite decreasing chain in S, there cannot exist any infinite $\mathcal{R}$-relabelling sequence. For our examples, the set S will be the set $\mathbb{N}^p$ where p is an integer and the ordering relation is defined by $(x_1,\ldots,x_p) >_p (y_1,\ldots,y_p)$ which means that there exists an integer j such that $x_1 = y_1,\ldots,x_{j-1} = y_{j-1}$, and $x_j > y_j$.

In order to prove the correctness of a graph relabelling system, that is the correctness of an algorithm encoded by such a system, it is useful to exhibit (i) some *invariant properties* associated with the system (by invariant property, we mean here a property of the graph labelling that is satisfied by the initial labelling and that is preserved by the application of every relabelling rule) and (ii) some properties of irreducible graphs. These properties generally allow to derive the correctness of the system. Let us illustrate these techniques for the computation of a spanning tree in a network with processor identities.

2.2 Distributed Computation of a Spanning Tree

Let G be a graph with n vertices. Each vertex has a unique identity; for simplicity we assume that it is a unique integer from 1 to n. Consider a labelling function $\lambda : V \cup E \longrightarrow [0..n]$, which is initialized to the identities for vertices and to 0 for edges. In order to compute a spanning tree, consider the algorithm encoded by the graph relabelling system $\mathcal{R}_1 = (L_1, I_1, P_1)$, defined by $L_1 = [0..n]$, $I_1 = [0..n]$, and $P_1 = \{R\}$ where R is the following relabelling rule:

$$R: \quad \overset{i}{\bullet}\!\!\overset{\alpha}{\text{———}}\!\!\overset{j}{\bullet} \quad\longrightarrow\quad \overset{i}{\bullet}\!\!\overset{i}{\text{———}}\!\!\overset{i}{\bullet} \quad ;\ \text{if } j < i$$

We prove now that this rewriting system computes a spanning tree. We show that it terminates and is correct.
Termination: Let f be the mapping from $\mathcal{G}_{L_1}$ to the set of natural integers $\mathbb{N}$ which associates with each L_1-labelled graph the sum $\sum_{\substack{v\in V,\\ \lambda(v)=k}} (n-k)$. Observing that this nonnegative number strictly decreases when we apply the relabelling rule R_1 we get that $(\mathbb{N}, >)$ is a noetherian order compatible with the system $\mathcal{R}_1$. Thus, we have the following lemma:

Lemma 1. *The relabelling system $\mathcal{R}_1$ is noetherian.*

Correctness: It is obtained by the following lemma:

Lemma 2. *Let $G(V,E,\lambda)$ be a connected labelled graph such that every vertex is labelled by its identity (unique integer of $[1,n]$ where n is the size of the graph), and every edge is labelled 0. Let $G'(V,E,\lambda')$ be a graph such that: $G(V,E,\lambda) \xrightarrow[\mathcal{R}_1]{*} G'(V,E,\lambda')$. Then the graph $G'(V',E',\lambda')$ satisfies:*

1. *All edges incident to an i-labelled vertex have labels lower or equal than i.*
2. *All vertices labelled n are connected by edges labelled n.*
3. *The subgraph induced by the i-labelled edges has no cycle ($i > 0$).*

4. If G' is an irreducible graph obtained from G, then all vertices are labelled n.

With all these properties, we have the following result:

Theorem 1. *The relabelling system $\mathcal{R}_1$ given above encodes a distributed algorithm generating a spanning tree.*

Note that, although the algorithm finishes, it does not have the *local detection property*. That is, at the end, every vertex is labelled n and hence computation has terminated, but no one can decide whether the global algorithm is finished. We shall present in the following subsections two other relabelling systems which have such property.

2.3 Sequential Computation of a Spanning Tree (using priority relation)

In this subsection, we will present a relabelling system to describe a distributed algorithm computing a spanning tree in a network with processor identities. We will show that it has the termination detection property: i.e. there is at least one vertex who detects the global termination.

Let G be a graph with n vertices. Each vertex has an unique identity; i.e. a unique number between 1 and n. Consider the labelling function λ, where $\lambda^{(V)} : V \to \{A, M, N, F\} \times [1..n]$ and $\lambda^{(E)} : E \to [0..n]$. Initially all edges are labelled 0 and each vertex is labelled (A, i) where i is its identity.

At each step of the computation, the (A, i)-labelled vertex, say u, will act as follows:

1. If u has a (X, j)-labelled neighbour v, where $j < i$ and $X \in \{A, M, N, F\}$, then u will activate this neighbour: u becomes marked (with label (M, i)), v becomes active (with label (A, i)) and the edge $\{u, v\}$ becomes i-labelled.
2. If u has a (X, j)-labelled neighbours with $j > i$ then u becomes neutral.
3. If u has no (X, j)-labelled neighbour, where $j < i$ and $X \in \{A, M, N, F\}$, and has a (unique) (M, i)-labelled neighbour w then u will reactivate this neighbour: u enters a final state (with label (F, i)) and w becomes (A, i)-labelled.

The computation stops as soon as none of the above computation rules may be applied (in that case, all the neighbours of the (A, n)-labelled vertex are (F, n)-labelled). The spanning tree is then given by the set of all n-labelled edges.

In order to reach a satisfactory expressive power, we introduce some local control mechanisms. These mechanisms allow us to restrict in some sense the applicability of relabelling rules. The first mechanism is obtained by introducing some *priority relation*, denoted $>$, on the set of relabelling rules. A relabelling rule R is applied if there exists no occurrence of a relabelling rule R' with $R' > R$ which intersects R (i.e. R and R' are applied on subgraphs having vertices in common).

For the computation of this subsection, it means that u will try to apply : first step 1 then step 2 and finally if it can not apply neither 1 nor 2 then step 3. The algorithm may be encoded by the graph relabelling system with priorities (PGRS for short) $\mathcal{R}_2 = (L_2, I_2, P_2, >)$ defined by $L_2 = \{\{A, M, N, F\} \times [1..n] \cup [0..n]\}$, $I_2 = \{\{A\} \times [1..n] \cup \{0\}\}$, $P_2 = \{R_1, R_2, R_3\}$ where R_1, R_2 and R_3 are the following relabelling rules:

$$R_1 : \overset{(A,i)}{\bullet} \overset{\alpha}{\text{———}} \overset{(X,j)}{\bullet} \longrightarrow \overset{(M,i)}{\bullet} \overset{i}{\text{———}} \overset{(A,i)}{\bullet} \quad ; \text{if } j < i \quad ; X \in \{A, M, N, F\}$$

$$R_2 : \overset{(A,i)}{\bullet} \overset{\alpha}{\text{———}} \overset{(Y,k)}{\bullet} \longrightarrow \overset{(N,i)}{\bullet} \overset{\alpha}{\text{———}} \overset{(Y,k)}{\bullet} \quad ; \text{if } i < k \quad ; Y \in \{A, M, N\}$$

$$R_3 : \overset{(M,i)}{\bullet} \overset{i}{\text{———}} \overset{(A,i)}{\bullet} \longrightarrow \overset{(A,i)}{\bullet} \overset{i}{\text{———}} \overset{(F,i)}{\bullet}$$

with the priority relation : $R_1, R_2 > R_3$.

The proofs of termination and of correctness are obtained by the following lemmas, where n is the size of G.

Lemma 3. *The relabelling system $\mathcal{R}_2$ is noetherian.*

Proof : This system is noetherian because executing a rewriting step implies that $(\sum_{\substack{v \in V, \\ \lambda(v) = (A,k)}} (n - k), |G|_A, |G|_M)$ is decreasing for $>_3$, where $|G|_A$ (resp. $|G|_M$) is the number of vertices with the label A (resp. M) in G.

Correctness is stated by the following lemma:

Lemma 4. *Let $G(V, E, \lambda)$ be a connected labelled graph such that every vertex has a unique identity ident belonging to $[1..n]$ and is labelled $(A, ident)$ and every edge is labelled 0. Let $G'(V, E, \lambda')$ be a graph such that: $G(V, E, \lambda) \xrightarrow[\mathcal{R}_2]{*} G'(V, E, \lambda')$ and let $X \in \{A, M, N, F\}$. Then the graph $G'(V, E, \lambda')$ satisfies:*

(I_1) *The labels of the edges incident to an (X, i)-labelled vertex are less or equal than i.*
(I_2) *There exists at most one vertex labelled (A, i), for a fixed $i > 0$.*
(I_3) *There exists at least one vertex with the label (X, j) neighbour of a (N, i)-labelled vertex with $j > i$.*
(I_4) *A (F, i)-labelled vertex has no (X, j)-labelled vertex as a neighbour, where $j \neq i$.*
(I_5) *All the (X, n)-labelled vertices are connected by n-labelled edges.*
(I_6) *The subgraph of $G(V, E)$ induced by the i-labelled edges has no cycle ($i > 0$).*
(I_7) *The (M, n)-labelled vertices form a simple path which edges are labelled n. Moreover, one of the endvertices of this path is connected to a vertex labelled (A, n) by an edge labelled n.*
(I_8) *Let G' be an irreducible graph obtained from G, only one vertex has (A, n) as label and all the others are labelled (F, n).*

Theorem 2. *The system $\mathcal{R}_2$ computes a spanning tree. This algorithm detects locally the global termination.*

With the properties $(I_4), (I_5), (I_6)$ and (I_7), we deduce that at the end, we have a spanning tree with (A, n) as root and all the other vertices are labelled with (F, n). The edges of this tree are labelled n. Indeed the (A, n)-labelled vertex having all neighbours labelled (F, n) knows that the algorithm is terminated.
Remark :
In this algorithm, many trees are computed distributively. However, each of these trees is computed in a sequential way.

2.4 Distributed Computation of a Spanning Tree with Local Detection of the Global Termination (using forbidden context)

Now, we present a relabelling system encoding an algorithm computing a spanning tree which has both properties: distribution and local detection of the termination. We use the forbidden context local control mechanism.

Let G be a graph with n vertices. Each vertex has a unique identity; i.e. a unique number between 1 and n. Consider the labelling function λ, where $\lambda^{(V)} : V \to \{A, A', F\} \times [1..n]$ and $\lambda^{(E)} : E \to [0..n]$. Initially all edges are labelled 0 and each vertex is labelled (A, i) where i is its identity.

The main idea is that the unique initially (A, n)-labelled vertex will keep its label until the end of the computation, while other activated vertices will be (A', n)-labelled. As soon as an (A', n)-labelled vertex is no longer "useful" for the growing of the tree, it will reach its final state (with label (F, n)). More precisely, we will use the following computation rules:

At each step of the computation, an active vertex (with label (A, i) or (A', i)), say u, will act as follows:

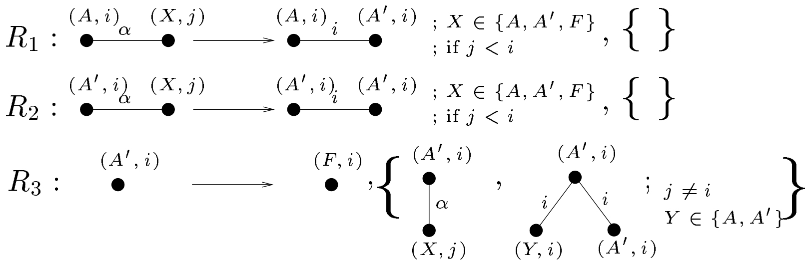
1. If u has a (X, j)-labelled neighbour v, where $j < i$ and $X \in \{A, A', F\}$, then u will activate this neighbour: u keeps its label, v becomes active (with label (A', i)) and the edge $\{u, v\}$ becomes i-labelled.
2. If u is (A', i)-labelled, has no (X, j)-labelled neighbour, where $j \neq i$ and $X \in \{A, A', F\}$, and is such that all its neighbours to which it is linked by an i-labelled edge except one of these neighbours are (F, i)-labelled, then u becomes (F, i)-labelled.

At any time, the subgraph induced by the n-labelled edges and the (A, n)- or (A', n)-labelled vertices is a tree. Intuitively speaking, the second rule means that the vertex u is a leaf in this tree.

Thus, this algorithm runs in two phases (that may overlap): in the first phase, the tree is growing until all vertices are reached; in the second phase, it will decrease (by loosing its leaves) until it is reduced to the initially (A, n)-labelled vertex. This vertex is then able to detect that the algorithm has terminated since all its neighbours are (F, n)-labelled.

Here we use another local control mechanism : *forbidden contexts.* A relabelling rule with forbidden contexts may be applied on some occurrence if and only if this occurrence is not included in an occurrence of some of its forbidden contexts.

The algorithm may be encoded by the graph rewriting system with forbidden context (FCGRS for short) $\mathcal{R}_3 = (L_3, I_3, P_3)$ defined by $L_3 = \{\{A, A', F\} \times [1..n] \cup [0..n]\}$, $I_3 = \{\{A\} \times [1..n] \cup \{0\}\}$, $P_3 = \{R_1, R_2, R_3\}$ where R_1, R_2 and R_3 are the following relabelling rules with forbidden contexts:



Rules R_1 and R_2 have not forbidden contexts: there is no restriction for the applications. Rule R_3 has 3 forbidden contexts (the subgraphs between brackets on the left-hand side). The unique vertex of the left-hand side of the rule R_3 is associated with the top vertex of its forbidden contexts. We will show that this rewriting system terminates and is correct. Termination is expressed by the lemma:

Lemma 5. *The rewriting system $\mathcal{R}_3$ defined above is noetherian.*

Proof: consider this couple $(\sum_{\substack{v\in V,\\ \lambda(v)=(X,k)}} (n-k), |G|_{A'})$, where $X \in \{A, A', F\}$. It is decreasing for $>_2$, so this system is noetherian.

Correctness is given by:

Lemma 6. *Let $G(V, E, \lambda)$ be a connected labelled graph such that every vertex v is labelled (A, i) and every edge is labelled 0, where $i \in [1..n]$ is the identity of v, n is the size of G. Let $G'(V, E, \lambda')$ be a graph such that: $G(V, E, \lambda) \xrightarrow[\mathcal{R}_3]{*} G'(V, E, \lambda')$ and let $X \in \{A, A', F\}$. Then the graph $G'(V, E, \lambda')$ satisfies:*

- (I_1) *The labels of the edges incident to an (X, i)-labelled vertex are less or equal than i.*
- (I_2) *There exists at most one vertex labelled (A, i), $i > 0$.*
- (I_3) *There is exactly one vertex with the label (A, n).*
- (I_4) *A (F, i)-labelled vertex has only (X, i)-labelled vertex as neighbours.*
- (I_5) *All the (X, n)-labelled vertices are connected by n-labelled edges.*
- (I_6) *The (A, n) and (A', n) labelled vertices are connected by n-labelled edges.*
- (I_7) *The subgraph of $G(V, E)$ induced by the i-labelled edges has no cycle.*
- (I_8) *Let G' be an irreducible graph obtained from G, only one vertex has (A, n) as label and all the others are labelled (F, n).*

Finally, we have:

Theorem 3. *The system $\mathcal{R}_3$ encodes a distributed algorithm which computes a spanning tree. Moreover, it has the termination detection property.*

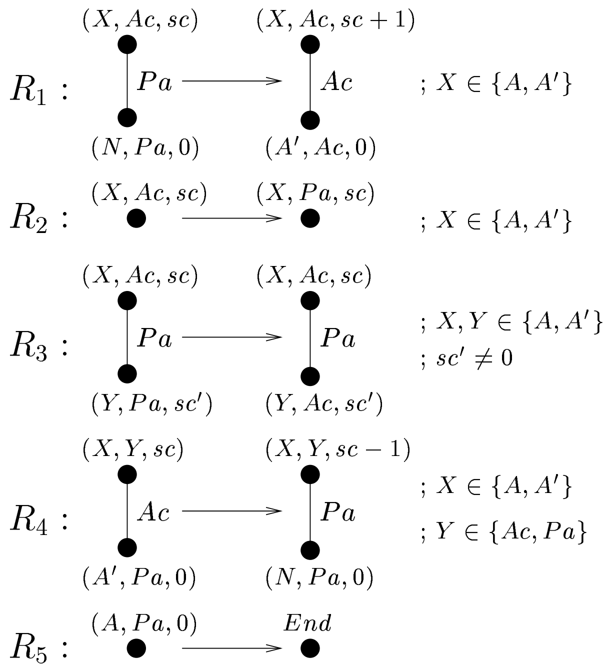
3 Termination Detection of a Graph Relabelling System

In this section, we give a solution to make a graph relabelling system detect locally the global termination. We use the well-known Dijkstra-Scholten [6] termination detection algorithm. More precisely, We show how to transform a graph relabelling system into another one which has the termination detection property. The method we present can be applied for any relabelling system which is initialized by exactly one vertex. We start by a description of the termination algorithm [6]. Then, we express it by a relabelling system, and finally we use it as a tool to detect locally the global termination.

3.1 The Dijkstra-Scholten Termination Detection Algorithm

Consider a network of processors modeled by a graph. Each processor is either in the state *active* or in the state *passive*. The processors communicate solely by messages (which are assumed to be transmitted instantaneously: synchronous message passing). Some of these messages are referred to as activation messages; they may set the receiver into an active state. An active processor maintains a count of the number of activations it has sent which have not yet been acknowledged. Initially, exactly one processor is active; then it activates some neighbours. The set of active nodes form a dynamic tree; where the root is the initial active node, and a node is the father of the nodes it activates. For each node, the transition from the active state to the passive state is an internal transition. A leaf node is deleted from the tree if it is passive and has no active descendants; i.e. it has received acknowledgments for the activation messages it has sent. In this case, it sends a signal to acknowledge the activation to its father who will decrease its own count of active sons. Note that the tree may grow and shrink dynamically. We will call this tree the *control tree*. The computation terminates when the root becomes passive and has no active descendants.

We give now a relabelling system to encode the Dijkstra-Scholten algorithm. To do so, we equip each vertex of the graph with a counter (say sc) and with a flag describing its activity status (say a value in $\{Ac, Pa\}$). Hence, a vertex will be labelled by a triple (X, Y, sc), where X is its label N, A, or A'; Y is its status activity, and sc the counter described above. An edge will be labelled by a label in $\{Ac, Pa\}$ indicating whether on not it belongs to the control tree. Initially, each edge is labelled Pa and each node is labelled $(N, Pa, 0)$; a distinguished node is labelled $(A, Ac, 0)$ which will be the root of the control tree. This node initializes the algorithm. Let $\mathcal{R}_4 = (L_4, I_4, P_4)$ be the relabeling system defined by $L_4 = \{A, A', N\} \times \{Ac, Pa\} \times [0..n] \cup \{Pa, Ac\}$, $I_4 = \{(A, Ac, 0) \cup (N, Pa, 0) \cup \{Pa\}\}$, and P_4 consisting of the following relabelling rules:



Note that R_3 allows an internal node of the control tree which is in a passive state to be activated by one of its active neighbours. This is a part of the Dijkstra-Scholten algorithm.

Theorem 4. *The relabelling system $\mathcal{R}_4$ encodes the Dijkstra-Scholten algorithm. If the label End appears, then each vertex is passive. Conversely, if each vertex is passive then eventually the label End will appear.*

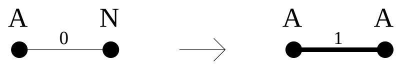
Proof: The proof follows from the invariants given below.

(I_1) sc+1 is the number of edges labelled Ac incident to the vertex labelled (X, Y, sc), where $X \in \{A', N\}$ and $Y \in \{Ac, Pa\}$. sc is the number of edges labelled Ac incident to the vertex labelled (A, Y, sc), where $Y \in \{Ac, Pa\}$.
(I_2) All edges incidents to a $(N, Pa, 0)$-labelled vertex are labelled Pa.
(I_3) A (A', Y, sc)-labelled vertex has at least one Ac-labelled incident edge, where $Y \in \{Ac, Pa\}$.
(I_4) The subgraph induced by (X, Y, sc)-labelled vertices is connected by Ac-labelled edges, where $X \in \{A, A'\}$ and $Y \in \{Ac, Pa\}$.
(I_5) The subgraph induced by the Ac-labelled edges has no cycle.

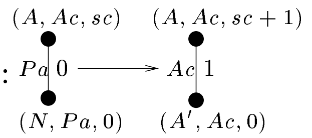
Remark. Encoding the Dijkstra-Scholten algorithm by a graph relabelling system makes it easy to read, to understand and particularly to prove in a unified and simple way as we showed for the previous algorithms. Moreover, expressing this algorithm in form of a graph relabelling system will be useful to apply the Dijkstra-Scholten termination technique to an algorithm encoded by a graph relabelling system. In fact, this turns out to be a combination of two graph relabelling systems as we will illustrate this idea by the following example.

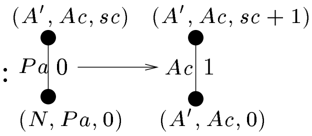
3.2 Adding the Dijkstra-Scholten Termination Detection Algorithm to a Graph Relabelling System

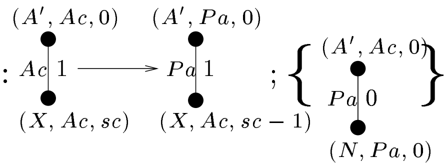
Now, we consider a relabelling system encoding a distributed algorithm which is initialized by one vertex. To detect termination by the previous technique, one has to embed the relabelling computation into the control tree. We have just to combine the graph relabelling system with the previous one. A node will be passive if it has locally terminated (no rule can be applied on its context). The rules of the relabelling system are mapped using the new triple labelling. In order to keep up with the control tree of the termination algorithm, a relabelling rule encoding local termination and acknowledgment must be added. As an illustrative example, consider the following relabelling system $\mathcal{R}_5 = (L_5, I_5, P_5)$ defined by $L_5 = \{N, A, 0, 1\}$, $I_5 = \{N, A, 0\}$, and $P_5 = \{R\}$ where R is the following relabelling rule:

$R:$ 

This relabelling system computes a spanning tree in an anonymous network, but does not have the termination detection property (similarly to the relabelling system of section 2.2). By combining it with the Dijkstra-Scholten algorithm, we obtain the following relabelling system with forbidden contexts $\mathcal{R}_6 = (L_6, I_6, P_6)$, defined by $L_6 = \{A, N\} \times \{Ac, Pa\} \times [0..n] \cup \{0, 1\} \times \{Ac, Pa\}$, $I_6 = \{(A, Ac, 0) \cup (N, Pa, 0) \cup \{(0, Pa)\}\}$, and P_6 consisting of the following relabelling rules:

$R_1:$ 

$R_2:$ 

$R_3:$  , $X \in \{A, A'\}$

Note that we give here a simplified system of the combination of the relabelling system $\mathcal{R}_5$ with that of the Dijkstra-Scholten relabelling system. It is a relabelling system with forbidden context. Rule R_1, which is similar to the original rule, maintains the count and the status of activated vertices. However, rule R_3 is added to collect information about nodes who locally terminated, and has received acknowledgments from all activated descendants ($sc = 0$). A node locally terminates if it has no neighbour labelled N which has not been yet active. Such a node, will be set in the passive status, and will send an acknowledgment to its father if its counter is null. In the relabelling rule, we just decrement the counter sc of the father. The whole computation terminates if the root becomes passive and has no active descendants.

Theorem 5. *The relabelling system $\mathcal{R}_6$ encodes a distributed algorithm which computes a spanning tree. This algorithm has the property of local detection of the termination.*

Proof:
The proof is similar to the previous systems. We give the invariants to prove correctness and termination. Let $G(V, E, \lambda)$ be a connected labelled graph such that a distinguished vertex is labelled $(A, Ac, 0)$, all other vertices are labelled $(N, Pa, 0)$ and every edge is labelled $(Pa, 0)$. Let $G'(V, E, \lambda')$ be a graph such that: $G(V, E, \lambda) \xrightarrow[\mathcal{R}]{*} G'(V, E, \lambda')$. Then the graph $G'(V, E, \lambda')$ satisfies:

1. All edges incident to a $(N, Pa, 0)$-labelled vertex are labelled $(0, Pa)$.
2. A (A, X, sc)-labelled vertex has at least $(1, Y)$-labelled incident edge, $(sc \neq 0,$ $X, Y \in \{Ac, Pa\})$.
3. The subgraph induced by the edges labelled $(1, X)$ has no cycle, $(X \in \{Ac, Pa\})$.
4. The (A, Ac, sc) and (A', Ac, sc') labelled vertices are connected by $(1, Ac)$-labelled edges.
5. More generally, all the (A, X, sc) and (A', Y, sc') labelled vertices are connected by $(1, Z)$-labelled edges, $(X, Y, Z \in \{Ac, Pa\})$.
6. A $(A, Pa, 0)$-labelled vertex has no $(N, Pa, 0)$-labelled neighbours.
7. If G' be an irreducible graph obtained from G then all vertices are $(A, Pa, 0)$-labelled.

The detection of termination of the algorithm occurs as soon as the counter (sc) of the root becomes zero (0).

Let us note that all the algorithms presented in this paper have been implemented and tested in ViSiDiA [3, 2]. ViSiDiA is a tool to simulate, visualize and animate distributed algorithms. It is based on graph relabelling systems.

References

1. D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on theory of computing*, pages 82–93, 1980.
2. M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001.
3. M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI, 2002.

4. M. Billaud, P. Lafon, Y. Métivier, and E. Sopena. Graph rewriting systems with priorities. *Lecture notes in computer science*, 411:94–106, 1989.
5. P. Boldi and S. Vigna. Computing anonymously with arbitrary knowledge. In *Proceedings of the 18th ACM Symposium on principles of distributed computing*, pages 181–188, 1999.
6. E.W. Dijkstra and C.S. Sholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
7. T. Kameda and M. Yamashita. Computing on anonymous networks: Part i - characterizing the solvable cases. *IEEE Transactions on parallel and distributed systems*, 7(1):69–89, 1996.
8. I. Litovsky and Y. Métivier. Computing trees with graph rewriting systems with priorities. *Tree automata and languages*, pages 115–139, 1992.
9. I. Litovsky and Y. Métivier. Computing with graph rewriting systems with priorities. *Theoret. Comput. Sci.*, 115:191–224, 1993.
10. I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.
11. Y. Métivier, M. Mosbah, and A. Sellami. Proving distributed algorithms by graph relabelling systems: Examples of trees in networks with processor identities. Technical report, LaBRI, 2002. In preparation.
12. M. Mosbah and A. Sellami. Visidia: A tool for the visualization and similation of distributed algorithms. *http://www.labri.fr/visidia/*.
13. P. Rosenstiehl, J.-R. Fiksel, and A. Holliger. Intelligent graphs. In R. Read, editor, *Graph theory and computing*, pages 219–265. Academic Press (New York), 1972.
14. G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, 2000.

Graph Computing Environment

Martin Faust

Research Center for Work, Environment and Technology (artec),
University of Bremen, faust@artec.uni-bremen.de,
WWW home page: http://www.e56.de/projects/gce/

Abstract. This paper introduces the concept of a unified graph computing environment based on the GRACE graph transformation language. An overview about design and implementation issues and how they are solved is given. One issue is that the implementation allows multiple approaches to be mixed in one program. At last a graphical user concept that allows to freely switch between different representations of a graph is presented.

1 Introduction

A graph computing environment is introduced, based on the language GRACE which is developed by an European research group. GRACE combines concepts of existing systems (PROGRES [5], AGG [6]) along with new features into a unique high level specification and graph transformation language.
GRACE is a theoretic concept with a sample implementation. In GRACEland [2] many of the features were implemented with a Virtual Reality frontend.
The aim of this project is to provide a convenient and easy to use interface for a common graph computing environment (GCE). GCE can be seen as the successor of GRACEland building a uniform platform for a graph rewriting language. The design process has been guided by the following aspects:

- Support of all features of GRACE,
- Approach independence,
- Graph class independence,
- Independence of any graphical interfaces and
- Availability under the GNU Public License.

The idea is, to create a collection of object-oriented software-components offering the base functionality under a model-view-controller (MVC) paradigm. This concept is adapted from the MVC concept that was introduced in Smalltalk [8] back in 1980s. The graph is the model. It is changed by a GRACE program which controlls the modifications with flow control elements. The view is the graphical user interface.

One advantage of this architecture is that multiple views (graphical frontends) of the same data are possible. The object-oriented approach makes it easier to reuse common components because of its inherent methods like subclassing.

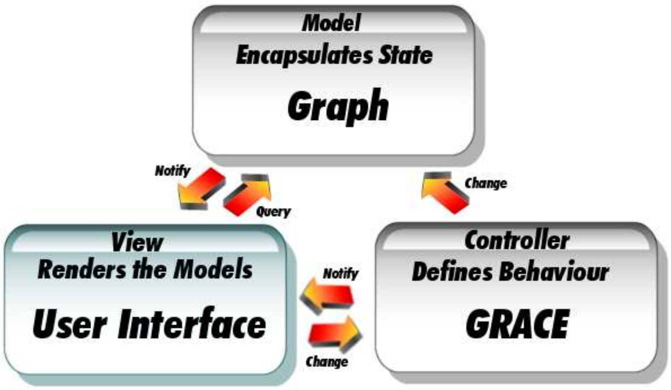


Fig. 1. The adapted Model-View-Controller paradigm

The graph computing environment is structured in three levels:

1. Abstract interface for graphs, rules, graph transformations,... ,
2. GRACE language working on top of the abstract interfaces and
3. User interface.

The first level defines basic classes and operations required from GRACE. Classes for nodes, links, graphs, rules and matches are building the abstract graph class description. Concrete graph classes (e.g. directed graphs) need to extend these classes and implement the virtual methods. Two operations can be identified as the common base needed from the second level:

1. Find a match m for the rule r under the approach a and
2. Apply rule r on graph g under a given match m and approach a.

These functions are bound to the selected approach, which defines the properties for the match and how a rule modifies a graph. The Single-Pushout (SPO) approach for example removes all dangling edges after the rule was applied. The functions are building the abstract approach class. The last functionality provided by this layer is the ability to load and save graphs and rules.

The second level consists of the language GRACE with its structuring and flow control elements. GRACE relies on the abstract operations and is not aware of any features of concrete graph classes. The lowest level of GRACE is the application of a rule in order to modify the graph. Methods for this operation are provided from the abstract graph class implementation. All other structs are high level elements that deal with structuring or controlling the flow.

The third level is the user interface for graph visualization and program editing. GCE is not restricted to a specific kind of user interface thus being flexible to meet the requirements of many users. The user interface depends on various aspects and for embedded systems it may be reduced to a minimum.

The rest of this paper is organized as follows. In chapter 2 the structure of the implementation is described. The GRACE language and it's embedding into GCE is outlined in chapter 3. Chapter 4 reveals some thoughts on graphical interfaces.

2 Implementation

In the introduction we already made some assumptions about the implementation which restrict us to object-oriented languages like C++ or Java. Java from Sun Microsystems was choosen as the programming language. Java's biggest asset is its capacity to run on various computers without recompiling. Also the good support for graphical user interfaces (using the Java Foundation Classes, Java2D/3D), database access, network and built-in security were the determining factor. The structure of the implementation reflects the introduced levels:

- Package **gce** defines the abstract interfaces
- Package **gce.grace** implements the GRACE transformation language
- Package **gceUI** implements one possible user interface

- Package **GRACEland** implements a GRACE runtime infrastructure
- Package **dg** implements a directed graph class
- Package **ex2** implements an expression language

2.1 Basic package

This package defines the abstract interfaces for graphs, nodes, links, matches, approaches, graph classes and input/output handling. I am not going into detail about the concrete implementation instead I will focus on two important design decisions.

The input/output class (gceIO) gives the user the possibility to load and save graphs and rules of any graph class. This is realized through the use of XML and the Document Object Model (DOM, [10]). gceIO loads XML files as a DOM tree into memory and extracts the required handler from the tree. The name of the handler is stored as an attribute of an element node. The Java class is loaded and creates the rule or graph from the tree.

This is a very powerful mechanism and has several advantages. First of all the handling and implementation of load and save functionality is straight forward. Many features can be integrated into the base class and doesn't need to be rewritten by each new graph class package. This concept is also open for extensions and/or modifications of the file format without changes to the basic functionality. An important feature is the possibility to embed a rule definition of arbitray graph class within a GRACE source code file (this will be discussed in chapter 3). However this is not a global file format [9] as currently developed by researchers. This file format will be supported in future releases.

Each rule has an application condition and a script which both may be empty. The condition controls the application of the rule and is evaluated at the search for a match. The script is used to compute values of attributes and is executed after the rule was applied. The language for these expressions is not defined by GCE. Instead I'm using an abstract interface to an expression handler. One possible implementation is outlined in chapter 2.3.

2.2 Graph classes

Graph classes are not part of the basic gce package. Each graph class has its own Java package. The name of the Java package is used by GRACE to identify the different graph classes.
To create a new graph class the user has to extend six classes only:

gceNode	gceLink	gceGraph
gceRule	gceApproach	gceIO

A graph class can have multiple approaches and not just one. Each approach must be implemented in its own Java class. Which one is used is decided by inspecting the name of the approach given in the GRACE module (see chapter 3). This does not conform to the GRACE specification but has practical and theoretical advantages. First of all it is easier to implement than the GRACE definition where the graph class is bound to the approach[1]. On the other hand it is now possible to easily mix different approaches for the same graph class in one GRACE program. A Single-Pushout module may be used to delete nodes because dangling edges are implicitly deleted and a Double-Pushout for the rest.

So far only directed graphs (dg) with a Double-Pushout are implemented.

2.3 Expression Language

As an example implementation of an expression handler a lightweight language (ex2) was created. The design of this programming language followed Wirth [1]. It should be self readable and contain only the most important features.
The elements of the expression language[2]:

- Local/External variables
- Assignments
- if E then S [else S] end
- forall v of {nodes, links}: {E, S} end
- exist v of {nodes, links}: {E, S} end
- break
- return E
- Operators and comparators

[1] In GCE the approach is bound to the graph class. Therefore the difference is not that big.
[2] E=Expression, S=Statements, v=Variable, a=Attribute

Local variables store intermediate results of boolean, integer, float, string, vector or handle types. Handles are references to nodes or links in graphs[3]. For the communication with the environment external variables are used. External variables can be named nodes/links or data given by the user. In an assignment a value is assigned to a variable or to an attribute of a handle. *Break* escapes from the current block.

The most important features are the *forall* and *exist* elements. They allow to test if all handles fulfill a requirement or if at least one handle with a special property exits. The forall and exist operator can be used as an expression (returning true or false) or as a statement. The context is defined by the rule and the match:

> *forall n of nodes: E end* is interpreted as
> $\forall x \in \{rule_{lhs}.nodes\} : n = match(x) \Rightarrow evaluate(E).$

In an application condition n is assigned to each node of the rule's left-hand side under consideration of the current match. Within a script n is assigned to each node of the rule's right-hand side.

If this language is used as an application condition the last statement is the *return* statement with a boolean result. In scripts the return statement is not used and omitted.

This lightweight language does not contain any advanced features like functions or classes. Nevertheless it allows to write quite complex expressions. A planned feature is to add the possibility to call Java methods. This offers the possibility of user interaction through dialogs, even more complex computations or using multimedia features like playing sounds. With external variables we have the possibility to define *template rules.* Templates are parameterized rules. With this construct we can write one rule that fit into various situations instead of writing one for each situation.

Example 1. A template rule for renaming nodes
A template rule for renaming the name of a node looks like this:

```
<rule name="rename">
  <node var="x"/>
  <script>x.name := newName;</script>
</rule>
```

Before the rule can be applied the external variable newName had to be defined, otherwise an exception is thrown. The definition is done when referencing the rule by adding a param attribute:

```
<rule name="rename" param="newName='unnamed'"/>
```

[3] Within the language it is not distinguishable between nodes and links by inspecting the variable type only. It seems that there is no case where someone would need this feature.

3 GRACE

The graph and rule centered language GRACE is implemented as a subpackage of gce. Because the language doesn't restrict itself to any kind of control elements this Java package provides only an abstract interface. Common to all GRACE language derivates are the module and transformation unit concept. They are described in detail in chapter 3.1. Figure 2 gives an overview about the compilation process and which packages are used. In order to execute a GRACE

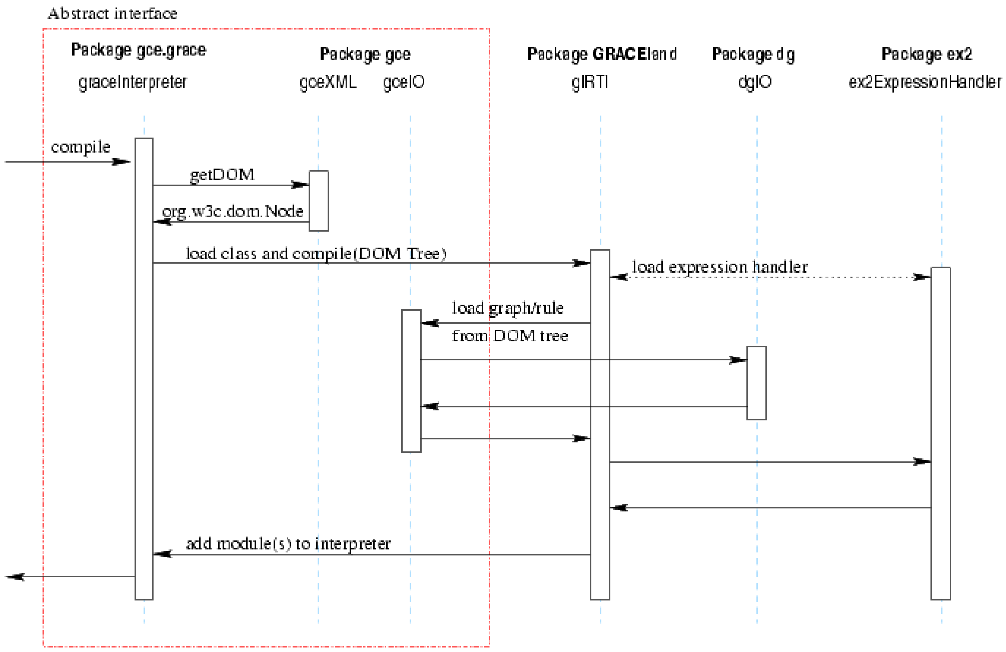


Fig. 2. Compilation of a GRACE program

program first an interpreter must be created. This is automaticly done by calling compile on the abstract graceInterpreter class. From the DOM tree the attribute interpreter(here GRACEland.glRTI) is extracted and the Java class is loaded. This class creates the runtime infrastructure. Afterwards the program can be executed.

The GRACE implementation is completly independent of any special expression/control languages and graph classes. An application can use different GRACE derivates without changing one line of code. All needed classes are loaded on demand.

3.1 GRACEland

GRACEland is a sample implementation of a GRACE interpreter and runtime environment. GRACEland programs are built from the following language elements:

Structure	Control	Computation[4]
transformation units	while E do S end	apply once {R, T}
modules	repeat S until E	apply n times {R, T}
	if E then S [else S] end	apply as long as possible {R, T}
		apply as long as one likes {R, T}

A *module* is a collection of private and public rules and/or transformation units. It provides functionality to other modules. In order to access the public elements the module must be imported first. Each module can have it's own graph class, approach and expression language. The mixture of different approaches and expression languages is supported but not between different graph classes. The current implementation supports a main transformation unit, which comparable to the "begin end." block in PASCAL. This main block is the entry point for the interpreter when executing a module.

Transformation units are abstractions from graph transformations. A unit has four elements: initial and terminal condition, local rules and a body.
Although the terminal condition still exists it is not used as an stop criteria as in [3]. Their understanding corresponds with the repeat–until concept. The semantic of a transformation unit regarding the terminal condition depends on the implementation. Here the terminal condition acts as a guard that tests if the graph has the expected property.

While, *repeat* and *if* are building the control conditions. Which expression language is used is specified at module level. In this paper I will use the expression language introduced in chapter 2.3 with one extension. The names of rules and transformation units may be used in an expression with the following semantic:

$$\text{rule}_{name} \Rightarrow \begin{cases} true & , \ \exists\text{morphism} \\ false & , \ otherwise \end{cases}$$

and

$$\text{transformation unit}_{name} \Rightarrow \begin{cases} true & , \ \text{initial condition is true} \\ false & , \ otherwise \end{cases}$$

With *apply* the graph is modified. The parameters are a set of rule and/or transformation unit names. Beside the shown modifiers that specifiy how often the command should be executed the user defines an order. The order influences how the interpreter selects the actual element. Selection is done randomly or sequentially where the first element is choosen that can be applied (see extension above). Random selection of a parameter provides a non predictable behaviour.

3.2 Example

The following example shows different aspects of the GCE design concept. GRACE-land and the forehand introduced ex2 language is used as the GRACE runtime

[4] R = Rule, T = Transformation Unit, as long as one likes = The interpreter rolls the dice to find out if he continues or stops.

infrastructure. This example provides an transformation unit that calculates the fibonacci number. The fibonacci number is defined by:

$$\begin{aligned} fib_0 = fib_1 &= 1 \\ fib_n &= fib_{n-1} + fib_{n-2}. \end{aligned} \tag{1}$$

The rule fExpand expands the fib(n) node into two nodes as defined in equation (1). fSum calculates the sum of two nodes. The other cases are handled implicitly by fExpand.

Example 2. A GRACE XML file for calculating Fibonacci numbers

```
<?xml version="1.0" encoding="utf-8"?>
<module name="fibonacci"              # The name of the module
        class="dg" approach="dgDPO"  # The graph class and approach
        expression="ex2"              # The expression handler
        grace="GRACEland.glRTI">      # The GRACE virtual machine

  <interface>      # Declaration of public elements
    <transformation_unit name="fib"/>
  </interface>

  <implementation>

    <comment>Calculates the fibonacci number</comment>
    <transformation_unit name="fib">
      <local>      # Two local rules
        <rule name="fExpand"> ... </rule>
        <rule name="fSum"> ... </rule>
      </local>
      <initial/>   # No initial condition
      <body>       # The calculation
        <apply mode="as long as possible"><rule name="fExpand"/></apply>
        <apply mode="as long as possible"><rule name="fSum"/></apply>
      </body>
      <terminal/>  # No terminal condition
    </transformation_unit>
  </implementation>
</module>
```

Each GRACE program starts with the definition of a module. The *module* tag is the only one that is common to all GRACE derivates. *class* defines the graph type that is used inside this program by specifing the name of its Java package. The approach to be used is given by *approach* and references a subclass of the Java graph package. *expression* specifies the expression language. All rules and GRACEland control elements of this module use this language. *grace* identifies the GRACE runtime environment that can interprete this progam. An instance of the interpreter compiles the file and creates the runtime structure (compare with figure 2).

Within the interface block all public elements are defined. These are accessible from other modules. The calculation of the fibonacci numbers is implemented by two locally defined rules that are used in the body of the transformation unit. GRACEland does not know how the rules can be parsed but knows that they are rules because of the *rule* tag. This information together with the specified graph type is passed to the base input/output class which calls the loader of the graph class. Rule tags outside the local section are references to unit local, module local or external rules.

4 Graphical User Interface

The editor is not part of the graph computing environment. It is up to the user to define an interface for editing GRACE programs. This decision is based on the assumption that representation issues are depending on the gaph class, the application and the users wishes. This makes it difficult if not impossible to find the one and only one graphical user interface. But nevertheless under certain constraints it is possible to have one common framework for the graphical user interface.

There are two kinds of editors that are comming into mind: a mixture of text and graphic elements and a complete graphical based approach. Figure 3 a) shows the visualization of example 2 being a mixture of text and images. The concept behind the framework is to leave the visualization of GRACE XML

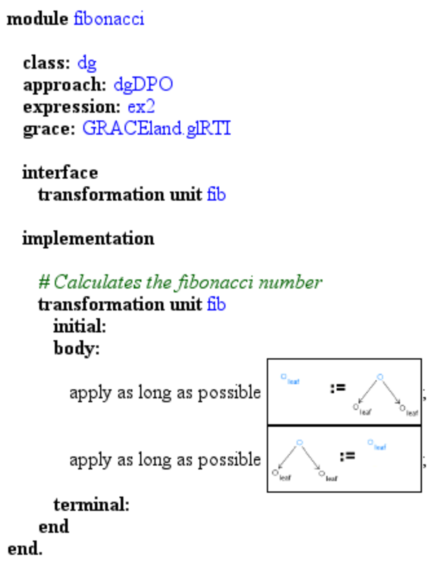


Fig. 3. Possible GRACE program visualizations

files up to individual implementations but provide common tools for editing and visualization of rules and graphs. Therefore only few components have to be rewritten. To deal with the different possible visualizations of a graph the concept of a design class is introduced. Figure 4 shows three different representations of directed graphs: If you compare the figures you will find out that the topologic

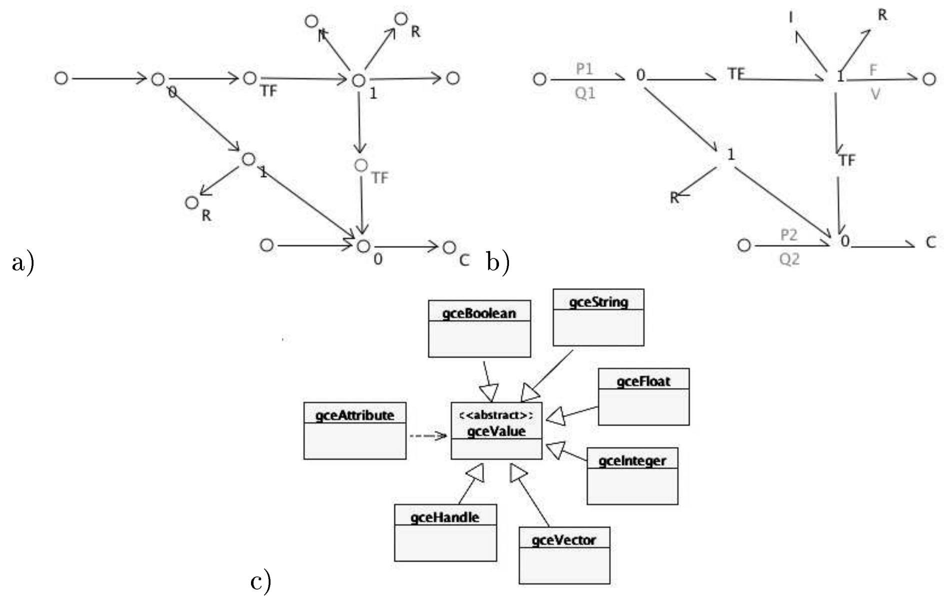


Fig. 4. Different design classes a) Standard b) Bond c) UML

information in a) and b) is the same but different attributes and drawing styles are used. A complete different drawing style is shown in c), where an UML drawing class was used. The idea is to associate a design with the graph by setting an attribute of the graph. It is up to the user which design he wants for a graph. According to the *design* attribute the corresponding drawing class is loaded. The editor is still the same only the drawing has changed.

What information is displayed is left to the special design classes. The standard design class recognizes only the label attribute. The bond drawing class recognizes the attributes label, effort, flow and causal. This concept of design classes is powerfull because the graph can be represented and visualized in a way, the user expects it.

5 Conclusion

With the graph computing environment a step towards a common platform for graph transformation is done. GCE does not try to create a new graph transformation theory but relies on a well known and proven theoretic framework. It tries to map the features from GRACE into a practical development environment.

GCE defines new concepts for the representation issues. The concept of different designs supports the expressiveness of graphs. The user is now able to freely choose the representation that is suitable for him. No mental transformation between two representations must be done.

The availibility under the GNU Public License gives researchers and developers the possibility to enhance and/or adapt the framework.

There are many things left that need to be examined in more detail, e.g. the possibilities and restrictions of mixing different approaches in one GRACE program. Also the notion of parallel and distributed computing need to be integrated into GCE.

Currently (at the time of writing) the implementation of the graph computing environment is in a pre-alpha state. All basic packages are done but the GRACE runtime infrastructure and the graphical user interface need some work. The implementation showed that the XML/DOM concept is efficient and helps to manage the handling of different graph classes.

For more information regarding GCE visit http://www.e56.de/projects/gce/.

References

1. Böszörményi, L., Gutknecht, J., Pomberger, G. (Eds.): The School of Niklaus Wirth. The Art of Simplicity. Heidelberg:dpunkt-Verlag (2000)
2. Faust, M.: GRACEland. Ein 3D-Editor und Interpreter für die graph- und regelbasierte Sprache GRACE. Diplomarbeit, University of Bremen (1998)
3. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D.,Schürr, Taentzer, G.: Graph transformation for specification and programming. Technical Report 7/96 University of Bremen (1996)
4. Burkhardt, R.: UML-Unified Modelling Language, Objektorientierte Modellierung für die Praxis. Addision-Wesley, Bonn (1997)
5. Schürr, A.: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems. Technical Report AIB 94-11, RWTH Aachen (1994)
6. Löwe, M., Beyer, M.: AGG - an implementation of algebraic graph rewriting. In Claude Kirchner (Ed.): Proc. Rewriting Techniques and Applications, Lecture Notes in Computer Science 690, 451-456 (1993)
7. Karnopp, D. C., Margolis, D. L., Rosenberg, R. C.: System Dynamics: A Unified Approach (2nd edition). John Wiley & Sons, New York (1990)
8. Krasner, G. E., Pope, S. T.: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. In Journal of Object Oriented Programming, Vol. 1 Nr. 3, 26-49, (1988)
9. Taentzer, G.: Towards Common Exchange Formats for Graphs and Graph Transformation Systems. Electronic Notes in Theoretical Computer Science http://www.elsevier.nl/locate/entcs (2001)
10. Document Object Model http://www.w3c.org/DOM/

Specifying Visual Languages with GenGED

Roswitha Bardohl, Karsten Ehrig, Claudia Ermel,
Anilda Qemali, Ingo Weinhold

Technische Universität Berlin
{rosi,karstene,lieske,aqemali,bonefish}@cs.tu-berlin.de

Abstract. This contribution gives an overview about the current concepts of GenGED, an environment for the visual definition of visual languages (VLs). From the visual definition, a VL specification is obtained that serves as a configuration of a VL-specific environment, i.e., the configuration is dependent on the parameters available in a VL specification: GenGED allows for the visual specification of syntax-directed editing, parsing, and simulation as well. In addition to these features, we show how to define an animation view for a certain VL model.
All GenGED features are based on the formal concepts of algebraic graph transformation and graphical constraint solving. Hence, we have a well-defined theory which serves as a basis for proper extensions of GenGED.
Keywords: visual languages, visual specification, editing, simulation, animation.

1 Introduction

The use of visual modeling and specification techniques today is indispensable in software system specification and development, so are corresponding visual environments. As the development of specific visual environments is expensive, generators for visual environments have gained importance, especially in the field of rapid prototyping. Most existing generators like DiaGen [10] rely on a textual specification of a visual language instead of a visual one. However, because of the at least two-dimensional character of visual representations the description by one-dimensional texts is not always adequate.

In this contribution we briefly present the current state of GenGED, developed at the TU Berlin, for visually specifying visual languages (VLs) and corresponding environments [1]. We start with a review on the underlying structure, namely alphabet and grammars based on graph transformation and graphical constraint solving. These structures form the basis for the specification of syntax and behavior of visual models (diagrams over a specific VL, such as statecharts, automata or Petri nets). The resulting specification is the basis for the configuration of a visual environment for (syntax-directed or free-hand) editing and simulation. The different components of GenGED are only loosely coupled to allow the user as much flexibility as possible. Thus, it is not necessary to define a parse grammar if the user wants to have a syntax-directed editor.

Syntax and parse grammars are needed for editing whereas simulation grammars describe the dynamic aspects of the specified system. The simulation of the system's behavior is defined by the means of the VL, e.g. the different states of the system are still given by diagrams over the VL, such as an active state of a statechart or an automaton, or a Petri net with an initial marking. Yet, in order to have an intuitive understanding of a model, it is even better to have an animation view which shows the dynamic behavior directly in the application domain. We sketch our ideas concerning an extension of GenGED towards allowing domain specific animation based on the formal simulation grammar.

The editing, parsing and simulation features are now implemented in the GenGED tool environment, whereas the extension concerning animation is still work in progress. All GenGED concepts are illustrated by the specification of a VL for automata comprising features for simulation and animation.

The paper is organized as follows: In Sect. 2 we briefly review the GenGED concepts which are illustrated by the specification of automata. The automata specification is extended in Sect. 3, where we discuss the specification of syntax-directed editing, parsing and simulation. In Sect. 4 we sketch our ideas for defining animation views of visual models.

2 Review of GenGED Concepts

GenGED is based on the well-defined concepts of algebraic graph transformation. Diagrams are represented by attributed graph structures covering the abstract syntax (the logical language elements) and the concrete syntax (the layout). A graph structure is given by disjoint sets, called *vertices* and unary operations from a source vertex to a target vertex, also called *edges*. Each vertex is typed over a *type graph* (the VL alphabet defining the vocabulary of a VL) such that the operations are structure preserving. The concrete syntax extends the abstract syntax by graphics defining the layout for each symbol type given by the abstract syntax. A graphical constraint satisfaction problem (CSP) over positions and sizes of the graphics defines the spatial relations between different symbols by restricting the scope of constraint variables. The CSP has to be solved by an adequate variable binding in each instance diagram over the alphabet. Thus, the CSP defines layout conditions for diagrams of a VL.

Using syntax-directed editing available in a VL-specific editor, a diagram is edited by applying the graph grammar rules to a given start diagram. The start diagram and the rules are part of the corresponding VL syntax grammar. In the following we illustrate the concepts *VL alphabet* and *VL grammar* by the specification of a VL for automata, our running example.

2.1 VL Alphabets

A VL alphabet establishes a type system for *symbols* (vertices) and *links* (edges) of a specific VL, i.e. it defines the vocabulary of a VL. The VL alphabet is represented by an algebraic graph structure signature and a constraint satisfaction problem [1].

A conceivable alphabet for automata is illustrated in Fig. 1. In the upper part of the figure, the abstract syntax of the alphabet is shown, namely the symbols State, Trans (short for *Transition*), and a Start (resp. Final) marking for a state. Both a State and a Trans symbol are enhanced with data attributes, namely a state name (short SN) and a transition label (short TL). We also introduce already the symbol Active which is used for the simulation (cf. Sect. 3.3). The links are indicated as arcs in the abstract syntax part of Fig. 1.

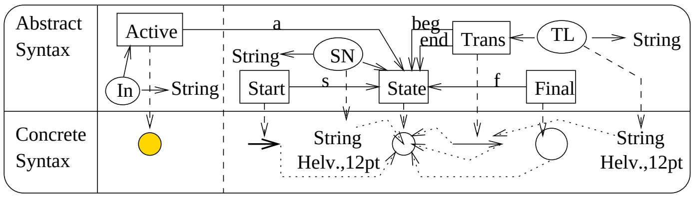


Fig. 1. VL alphabet for automata.

In addition to logical vertex attributes, symbol graphics are represented as a further kind of attributes. In Fig. 1, e.g., the graphic for the symbol type State is given by a circle. Thus, each State symbol (instance of the State type) in a diagram is represented by a circle. In general, the position and size of all instances occurring in a diagram depend on graphical constraints (illustrated by dotted arrows in Fig. 1). The condition that each start and end point of a transition arc must touch the boundary of a state circle is one example for a layout condition defined by constraints in our alphabet for automata.

Fig. 2 depicts the abstract and the concrete syntax of an instance over the automata alphabet modeling a process of the well-known producer/consumer system.

The states of the automaton represent the states of the system: A producer can be idle or busy and deliver a product to a buffer. A consumer can order a product, remove it from the buffer and consume it. The automaton verifies strings of the form $(pdro)^n pdrc$ where each character corresponds to a possible state transition (p: produce, d: deliver, r: remove, o: order, c: consume).

2.2 VL Grammars

Given a VL alphabet, a VL grammar over the VL alphabet consists of a start diagram and a set of rules. Usually, a rule consists of a rule name, optionally a set of parameters, and two graphs, namely a left-hand side (LHS or L) and a right-hand side (RHS or R), which are combined via a rule morphism (a graph structure morphism on the abstract syntax level). Moreover, a rule may be extended by negative application conditions (NACs) and attribute conditions that are boolean expressions over variables and parameters of a rule.

Graph transformation defines a rule-based manipulation of graphs. In GenGED we follow the Single-Pushout (SPO) approach to graph grammars [11] as well as

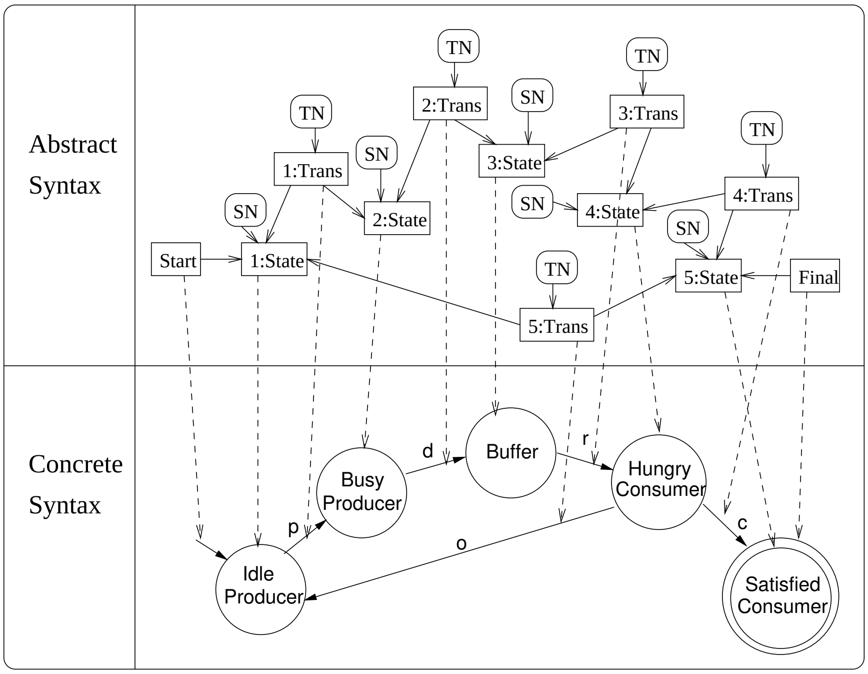


Fig. 2. Automaton modeling the Producer/Consumer system

we support the *dangling condition* well-known from the Double-Pushout (DPO) approach [6] [1]. The application of a rule r to a graph G (*derivation*) requires a mapping (total graph structure morphism) from the abstract syntax level of the rule's LHS to the abstract syntax level of this graph G. Due to the derivation result, the corresponding graphical attributes and constraints are instantiated from the alphabet. The positions and sizes of the graphical objects are calculated by a constraint solver (cf. [9]).

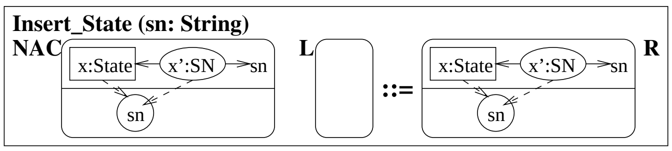


Fig. 3. Syntax-directed editing rule supporting the insertion of a state symbol.

Fig. 3 illustrates a syntax rule for the insertion of a State symbol. This rule contains a rule parameter, namely a state name indicated by the variable sn of type String. The left-hand side L of this rule is empty, i.e., nothing is required for

[1] In contrast to the SPO approach where all dangling edges are deleted implicitly, the dangling condition of the DPO approach forbids the rule application if the transformed graph H contains dangling edges.

applying the rule. By the right-hand side R a state symbol is generated together with a state name. The NAC states that the state names have to be unique in a diagram.

3 VL Specification

In GenGED algebraic graph transformation and graphical constraint solving techniques are combined to support the definition of VL specifications which configure a VL-specific visual environment. Each VL specification consists of a VL alphabet and some kinds of grammars, respectively specifications; cf. Fig. 4.

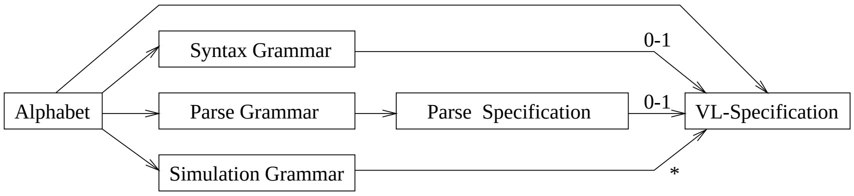


Fig. 4. Workflow for the visual definition of a VL specification.

If only a syntax grammar is in the focus of a VL specification, it should be defined in a way such that it expresses the correct syntax of the VL. Moreover, it should not cover only language-generating rules but additionally language-manipulating rules for comprehensive syntax-directed editing. Unfortunately, such a rule set can be very detailed and large, and an end user [2] may be irritated because of many rules doing more or less the same. Therefore, free-hand rather than syntax-directed editing is conceivable in a VL-specific visual environment. In this case a parse grammar should be available. In order to define complex VLs (like Statecharts in [2]) we propose the combination of a simple syntax grammar together with a parse grammar. A parse grammar may be extended by the definition of a layering function and a critical pair analysis in order to optimize the parsing process. The parse grammar together with these extensions result in a parse specification. Similar to the syntax definition via syntax grammar and parse grammar, the simulation is defined by a simulation grammar.

3.1 Syntax Grammar

Each rule of the syntax grammar provides an editing command. The grammar's start diagram serves as a template for new diagrams to be created. Usually, the editing process begins with an empty start graph. The syntax rules presented in

[2] We distinguish two kinds of users, namely users defining a VL (*language designer*), and those who use a VL specific environment (*end user*).

Fig. 5 are language generating; the rules for modification and deletion of elements work analogously. Fig. 5 illustrates three of four rules of the syntax grammar for the automata example – the first one, Insert_State(), was already given in Fig. 3. Insert_Transition() inserts a labeled transition between two states, Mark_Start() and Mark_Final() make a state the start respectively a final state by attaching the corresponding symbol. The Mark_Final() rule contains an NAC to avoid attaching the Final symbol twice. A similar NAC could be added to Mark_Start() as well, but since our parsing grammar covers this case, we omit it here.

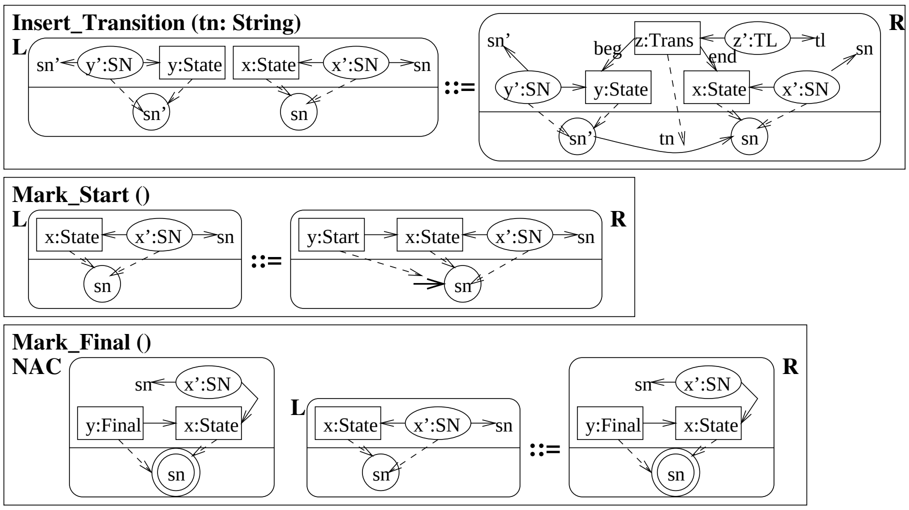


Fig. 5. Syntax grammar for automata.

3.2 Parse Grammar

Using the rules of the parse grammar the parser tries to reduce a given diagram to the grammar's stop diagram. If such a derivation exists then the diagram is accepted, otherwise rejected. An optional layering function assigns an integer number (a layer) to each rule. The parsing algorithm considers only those derivations consisting of rule applications of ascending order. Another optional feature, the critical pair analysis, is used to optimize the parsing process. Usually each possible order of applications of rules (of the same layer) has to be considered. Therefore for each pair of rules it is analyzed whether or not their matches might interfere with each other. In the latter case their application order does not play a role and thus only one single (arbitrary) order needs to be checked.

The parse grammar for the automata example (see Fig. 6) does not require layering. It is quite simple, since the only condition to be checked is whether the given automaton has exactly one start state. The rules Remove_Transition(), Unmark_Final() and Remove_State() resemble the respective inverted syntax rules (not requiring any NACs, if the dangling condition is used). They reduce the

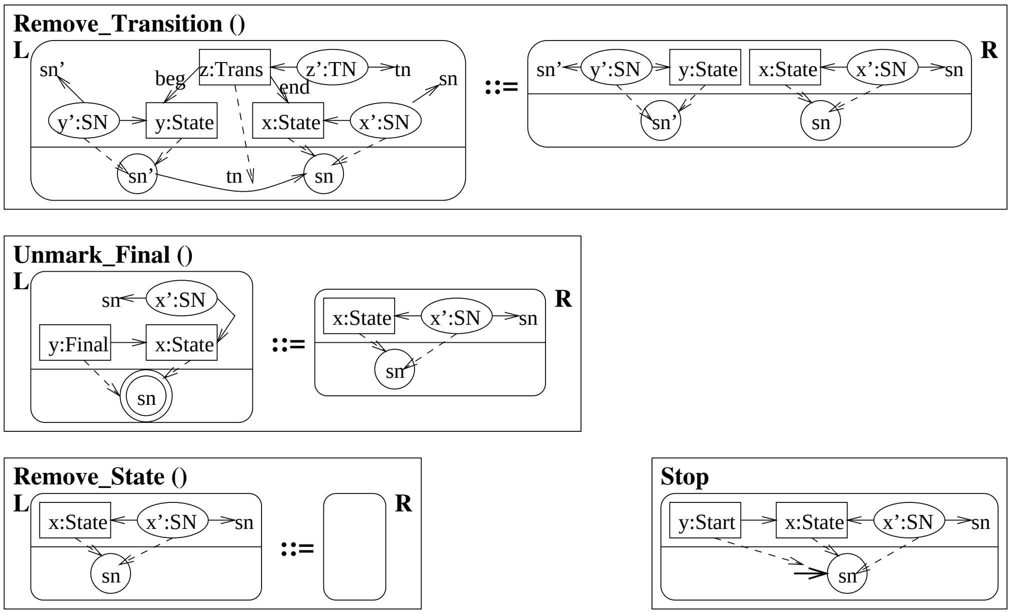


Fig. 6. Parse grammar for automata.

diagram by removing the Trans, Final and State symbols. The only symbols that cannot be removed are State symbols with attached Start symbols. Thus what should remain reducing a correct diagram is the start state, just as given by the grammar's stop diagram.

3.3 Simulation Grammar

In order to visualize which state is the active one, the automata alphabet (Fig. 1) contains an Active symbol, a colored circle. We want to simulate how an automaton reacts to a given input string. Therefore in each step the Active mark should move to the succeeding state. The Active symbol has an In attribute which contains the remainder of the input string still to be processed. Since we do not exclude nondeterministic automata, more than one transition might be triggered at a time. Fig. 7 illustrates the simulation grammar for automata.

The simulation grammar contains only two rules, Init() and Trigger_Transition(). The first one adds an Active symbol to the start state of a diagram. The rule parameter in, the input string, is stored as In attribute of the Active symbol. The second rule moves the Active symbol from the source to the target state of a transition. The attribute condition ensures that the transition label is indeed a prefix of the remaining input string. After the rule application the prefix has been removed.

Given the string to be processed, the simulation grammar calculates a state at which the automaton may terminate, if Init() is applied exactly once and then Trigger_Transition() is applied as long as possible.

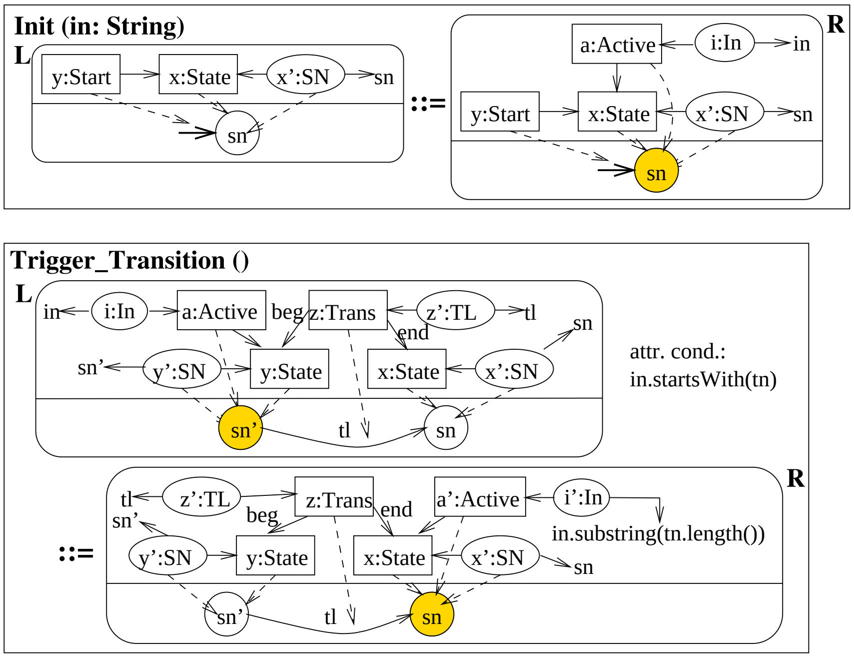


Fig. 7. Simulation grammar for automata.

The concepts of VL specification presented so far are the basis to generate a VL environment supporting editing and simulating specific diagrams (e.g. automata). In general, simulation grammars capture the behavior of formal visual models whose VLs allow the description of dynamic state transitions (like e.g. Petri nets, Statecharts or automata). Each simulation step (the application of a simulation rule) models a state transition.

However, the simulation process is still visualized by sequences of formal VL diagrams, i.e. specific automata or Statecharts are shown. In order to support an intuitive understanding of system behavior, especially for non-experts in the specific formal model, it is desirable to have a layout of the model in the application domain.

4 Defining Animation Views for Visual Models

In order to support an intuitive understanding of system behavior, especially for non-experts in the specific formal model, it is desirable to have a layout of the model in the application domain. In the GenGED approach it is possible to define a relationship between the formal system model and a corresponding layout of the model as icons from the application domain. Such an *animation view* directly shows the states and dynamic changes of the system.

Fig. 8 a) shows a certain state of the producer/consumer automaton from Fig. 2. The Buffer is highlighted as active here. For the animation view we choose the application domain of a kitchen. Producing is visualized as baking and consuming as eating cakes[3]. Fig. 8 b) shows a snapshot of the active system state in the animation view where the consumer has removed the product from the buffer.

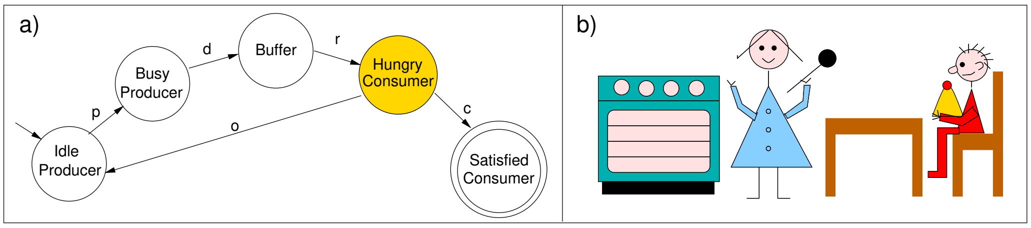


Fig. 8. Automaton and Animation View Snapshot of a state of the Producer/Consumer System

Within the GenGED framework, we suggest a generic approach how to visualize the animation of a system based on a VL specification, a VL model (VL diagrams for the different states of a system), and a VL simulation grammar. The transformation from the layout of the formal model to the layout of the animation view is called *view transformation*. Naturally this view transformation is formalized as a visual grammar based on the VL alphabet which is extended by the new graphics and constraints needed for the domain-specific layout. The simulation rules are transformed into animation rules for the animation view defining the state transitions in the new animation layout. We enforce compatibility between animation and simulation rules by applying the view transformation rules to the LHS and the RHS of each simulation rule instance.

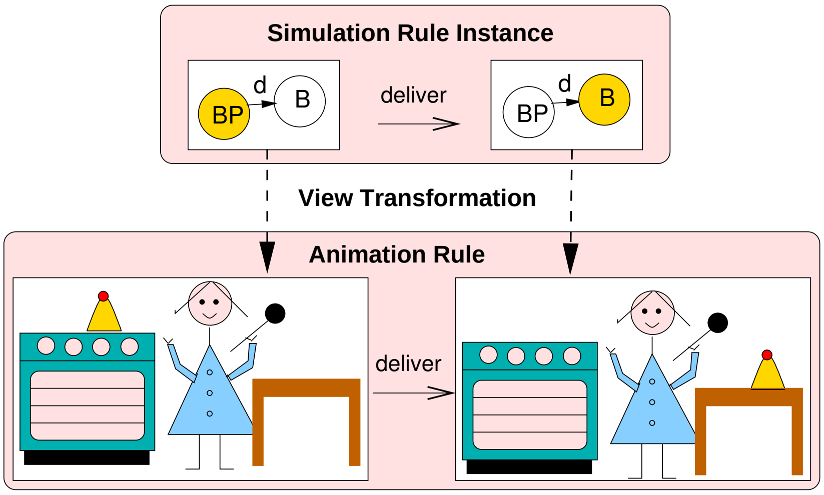


Fig. 9. View Transformation from Simulation to Animation View

[3] This is not meant to be discriminating: Also men bake cakes!

The simulation rule in the upper part of Fig. 9 models the state transition d (for deliver) from the state Busy Producer to the state Buffer by highlighting the current state. The animation rule in the lower part of Fig. 9 shows the same state transition in the application domain oriented layout. The dashed arrows from the simulation rule to the animation rule indicate the formal view transformation between both views.

The implementation of these concepts in the GenGED environment is work in progress and sketched in [7].

5 Conclusion

In [1] GenGED is proposed for the visual definition of visual languages (VLs) and graphical editors supporting syntax-directing editing. Meanwhile, GenGED has been extended in different fields as presented in this contribution. Not only syntax-directed editing may be defined visually but a parse specification and a simulation grammar as well. These specifications based on algebraic graph transformation allow comprehensive editing and analysis as well as they support the visualization of behavioral aspects of VL models.

Apart from the animation concepts, all the proposed concepts are implemented in the GenGED environment (see http://tfs.cs.tu-berlin.de/genged). The development of the animation approach is also joint work in the area of applying graph transformation techniques to Petri nets [5]. More details can be found in [3, 7] where different types of Petri nets (i.e. Elementary nets, Place/Transition nets and Algebraic High-Level nets) have been specified as VLs in GenGED .

Future directions concern the development of animation modules for different views of system behavior [4]. The specification of model evolution [8, 12] based on two different VLs modeling two layers of abstraction (architecture and components) is an example for the integration of views in one specification. Both VLs are coupled via distinguished (abstract) vertices. This may serve as basis for integrating several VLs in a way that it is possible to handle different kinds of views which are standard practice in the software specification process.

Acknowledgements The research is partially supported by the German Research Council (DFG), and the projects APPLIGRAPH (ESPRIT Basic Research WG), GRAPHIT (CNPq and DLR), and the joint research project "DFG-Forschergruppe Petri Net Technology". Many thanks also to our anonymous referees for valuable comments.

References

1. R. Bardohl. GenGEd – *Visual Definition of Visual Languages based on Algebraic Graph Transformation.* Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999.
2. R. Bardohl and C. Ermel. Visual Specification and Parsing of a Statechart Variant using GenGEd. In *Proc. Symposium on Visual Languages and Formal Methods (VLFM'01)*, Stresa, Italy, September 5–7 2001.

3. R. Bardohl, C. Ermel and H. Ehrig. Generic Description of Syntax, Behavior and Animation of Visual Models using GenGED. Techn. Report No. 2001/19, ISSN 1436-9915, TU Berlin, 2001.
4. R. Bardohl, C. Ermel, and L. Ribeiro. A Modular Approch to Animation of Simulation Models. In *Proc. 14^{th} Brazilian Symposium on Software Engineering*, Joao Pessoa, Brazil, October 2000.
5. B. Braatz, K. Ehrig, K. Hoffmann, J. Padberg, and M. Urbášek. Application of Graph Transformation Techniques to the Area of Petri Nets. In H.-J. Kreowski, editor, *Proc. AGT 2002: APPLIGRAPH Workshop on Applied Graph Transformation*, 2002. To appear.
6. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.
7. C. Ermel, R. Bardohl, and H. Ehrig. Specification and Implementation of Animation Views for Petri Nets. In Weber et al. [13], pages 75–92.
8. C. Ermel, R. Bardohl, and J. Padberg. Visual Design of Software Architecture and Evolution based on Graph Transformation. In *Int. Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01), ENTCS Vol. 44, No. 4, 2001.*
9. P. Griebel. *Paralleles Lösen von grafischen Constraints.* PhD thesis, University of Paderborn, Germany, February 1996.
10. O. Köth and M. Minas. Generating Diagram Editors Providing Free-Hand Editing as well as Syntax-Directed Editing. In H. Ehrig and G. Taentzer, editors, *Proc. GRATRA'2000 - Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 32–39. TU Berlin, March 25-27 2000.
11. M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M. Sleep, M. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.
12. J. Padberg, C. Ermel, and R. Bardohl. Rule-Based and Visual Model Evolution using GenGEd. In *Proc. Satellite Workshops of 27th Int. Coll. on Automata, Languages, and Programming (ICALP'2000)*, pages 467–475, Geneva, Switzerland, 2000. Carleton Scientific, Canada.
13. H. Weber, H. Ehrig, and W. Reisig, editors. *2nd Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin, Germany, Sept. 2001. Research Group "Petri Net Technology", Fraunhofer Gesellschaft ISST.

Graph technology applied to editing structured natural-language documents

Felix H. Gatzemeier,
Oliver Meyer*

Lehrstuhl für Informatik III, RWTH Aachen
{fxg,omeyer}@cs.rwth-aachen.de

1 Introduction: Conceptual Authoring Support

This is a report on our experiences of applying graph technology in the application area of creating and maintaining structured digital documents during the last four years.

Our aim is to contribute to the development of tools for authors of documents with an inherent content structure, such as scientific articles or textbooks, that help building and maintaining that structure. One goal of structural integrity is, for example, referring to terms only after they have been defined and then referring to them consistently with the definition. Another goal is for each section and subsection to play a clear and discernable role in the document.

The data from the document we need to help achieve these goals are the location and content of definitions, as well as the location of references. More data may be used to offer more functionality, for example the intended style of a presentation (inductive or deductive as prime alternatives), which would have to be consistent with the relationships of terms and the order of their occurrences in the text.

We call the general structure of content in the document the *content structure*. In the aforementioned form, it consists of concepts that may have various relationships among each other, for example one being a part of another, or being required for the understanding of another concept. Elements of the content structure may be connected to the visible parts of the document, for example as being defined or referred to. We call these visible parts the *presentation*. This corresponds to the level current word processors regard a document: a hierarchy of sections with typographically marked-up text and other media elements. We call these constituent parts the *document hierarchy* of *division*s on the one hand and the *media content* on the other.

Our tools provide means to construct documents containing content structure as well as the presentation and to check them according to rules of readable structuring. They are to provide technology, not policy in the sense that the

* This work has been funded by the noneDeutsche Forschungsgemeinschaft (DFG) in its "Schwerpunktprogramm V3D2" (noneVerteilte Verarbeitung und Vermittlung digitaler Dokumente, Distributed Processing and Exchange of Digital Documents), http://www.cg.cs.tu-bs.de/dfgspp.VVVDD and Grant No. NA 134/8-1.

author should be free to work with or without content-structural support solely at his discretion.

Divisions, concepts, definitions, references and concept relationships naturally form a graph. We have therefore used a graph database and a graph-based programming language to develop our tools. In doing so, we have approached the same problem with different emphasises and different tools of implementation, giving rise to a comparison of the results and implementations.

In this article, we first present the two projects in chronological order. This begins with `rwe`, an early prototype implemented in an imperative programming language and storing the document in a graph database. The follow-up is CHASID, which is implemented chiefly as an executable PROGRES specification. In each presentation, the functionality, implementation structure, noticeable technicalities and some results are mentioned. The last section compares the results and implementations before concluding with some notes on further plans.

2 `rwe`: Graphs and imperative programming

The first prototype, *rwe* (for *R*eader-*W*riter *E*nvironment) [2], aims at being an environment for authors and readers to add structure to an existing document. In the implementation, this existing document may be imported as media content from FrameMaker text files in MIF (Maker Interchange Format). The author *marks up* up parts of the imported media content (for example, paragraphs or chapters, but also arbitrary discontinuous parts) as playing a structural role, such as defining something or being defined. From these basic markups more comprehensive structures like definitions or Toulmin-style arguments [13] are built. In doing this, the author directly manipulates the graph structure. This graph is displayed using boxes and lines connected to the text columns.

Fig. 1 shows a screenshot of `rwe` displaying a cut-out of such a graph. The term "Software crisis" is marked up as being "Introductorily Occurring", which means that this is an occurrence of the term in which it is introduced. That mark-up has then been used in creating a "Definition". Other nodes represent the document hierarchy.

`rwe` allows to export HTML-files that use definitions to create an hyperlinked index. A branch of the development features variants of a document to be edited together. *Variant Document*s are nodes in a tree of descendants, with changes from parents being promoted to children, unless the parts of the document they affect are redefined there.

The allowed documents are constrained by the graph schema. It defines the assignable node types, subtype relationships and edge cardinalities. For example, a markup node such as the "Introductorily Occurring" above, must refer to a media content node.

The author may, however, deviate from the content structure's graph schema in controlled ways. When inserting a node, its edge cardinalities are satisfied by inserting *placeholder* nodes, which may later be instantiated or merged with real

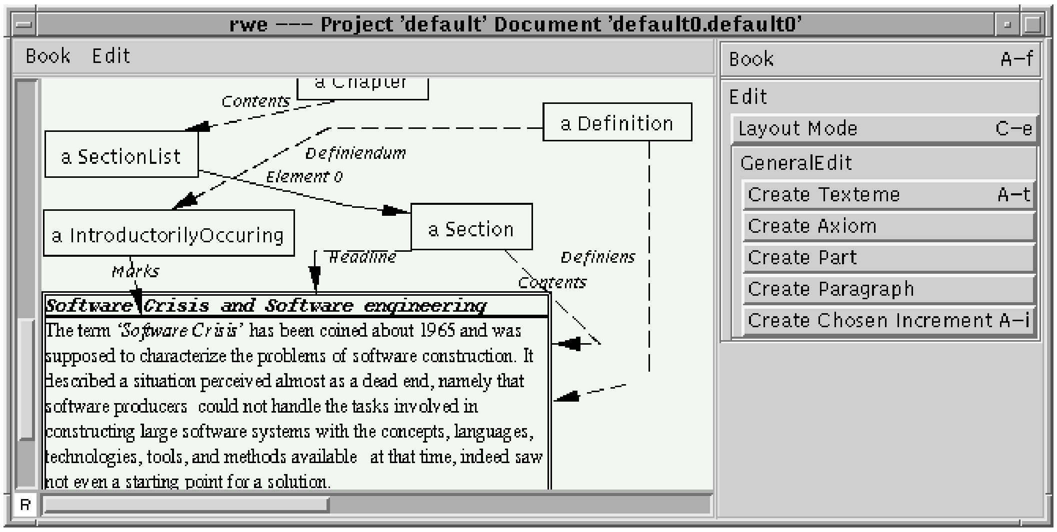


Fig. 1. Screenshot of `rwe`.

nodes. The author may also extend the schema between runs to accommodate personal structuring needs or preferences.

2.1 Implementation

The skill profile of the project participants led to the decision to implement `rwe` based on the IPSEN architecture [8]. The graph part was largely constrained to the persistent data storage in GRAS3 [1], a graph database implemented in Modula-3.

We used GRAS3's graph types directly to define the schema of `rwe`'s documents. Every node type defined in the content structure part of the graph database schema is available to the author for structuring his document. Most restrictions on the content structure are left for the database to check. Consequently, edge cardinalities and node types are the primary structural constraints checked by this prototype. On the other hand, adding new markup types or specialised content structures require only an extension of the declarative graph schema. This is possible between runs of `rwe`. The existing and newly added node and edge types are then available to the author through a *`GeneralEditTool`*.

This `GeneralEditTool` offers commands to insert arbitrary allowed edges by querying the schema for all edge types that may be inserted between the currently selected nodes. Nodes are created independent of the selection, but with the obligatory context (as determined by querying the schema) inserted automatically in the form of placeholders. Placeholders may be merged at any time with existing nodes using the *`PlaceholdingTool`*, if edge cardinalities and type constraints permit. All edges occurring at the placeholder are redirected to the remaining node.

To implement tool support for special structures, like those created by the MIF import or used by the HTML export, some parts of the schema are fixed as the compiled code relies on them. We call these parts the *well-known schema*.

We designed a domain-specific schema containing 44 concrete node types and 19 additional abstract classes. These schema entities include types for the document hierarchy like PARAGRAPH or SECTION, technical types like CHARACTERFORMAT or INTERNALLISTINCREMENT as well as the above-mentioned structural elements like ARGUMENT or DEFINITION. 21 of these schema entities constitute the well-known schema.

As the author directly manipulates the graph database's schema, consistency between the schema demanded by compiled code and the schema provided at run-time is of great concern to us. We generate type-safe encapsulation classes for the well-known schema. These classes check the existence of required node types at start-up time. Consistency between the implementation and the well-known schema is then provided by Modula-3's compile time checks.

Only very limited editing operations have been implemented for the media content. It is also stored in GRAS3. This allows direct references to it from the content structure through regular graph edges. The *reference problem* of referring to already deleted parts of the media content from the content structure is therefore avoided.

2.2 Results

Maintaining the entire document, including the media content, in a graph database and implementing the presentation and all editing operations in a new application proved to be even beyond research prototype feasibility. To begin with, the presentation response time was simply insufficient for interactive work. Media support needed to be extended to at least cover figures if not general multi-media content. Missing convenient editing operations for media content, powerful constraint checks for the content structure as well as parameterised extended export functionality to various formats, make `rwe` not the authoring environment it tried to be. It has not been used for any serious modelling activity. In our second approach we therefore concentrated solely on the content structure, leaving presentation issues to established applications (see section 3).

As for imperative graph-based implementation, some general experiences have been made:

- Generating type-safe interfaces from a subset of the schema description has proven to be a useful compromise between compile-time checks and runtime flexibility.
- The limited scope of cardinality-based schemata necessitates maintaining consistency of subgraphs in the application. Another layer of specialised generated wrapper classes can provide this for frequently used patterns like lists.
- The generic operations of the `GeneralEditTool` and `PlaceholdingTool` are fairly straightforward to implement.

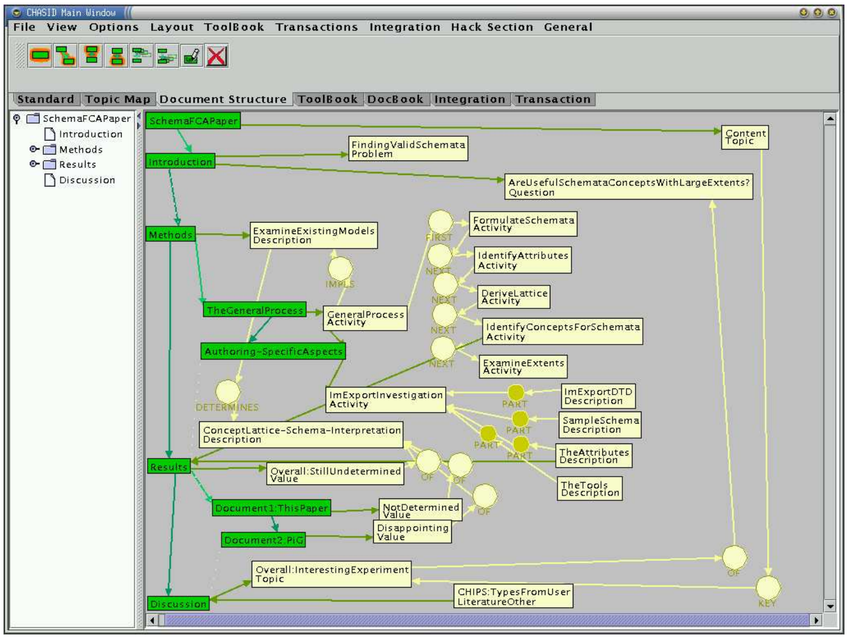


Fig. 2. A main window of a running Chasid prototype

– Subgraph matching, closure calculation and similar graph operations are frequently inadequately described in textual imperative languages.

3 Chasid: Specifications and Integration

Chasid (*c*onsistent *h*igh-level *a*uthoring and *s*tudying with *i*ntegrated *d*ocuments, [3, 4]) addresses `rwe`'s usability problem by working as an extension to existing authoring environments (called *conventional applications* here)[1]. In this setup, authors create the media content in the environment they know, while the Chasid extensions keep track of the content structure. The document hierarchy is used as the interface between the conventional document and the content structure. The document the author edits now has, technically, two parts: one that the conventional application maintains, called the *conventional document*, and the other one in the extension, called the *document graph*.

The screenshot in figure 2 shows the extension while the author works on the structure of an article about an analysis process of content structures. On the right, the document hierarchy is presented in a tree view, repeated in the document graph view with the nodes with a dark background. Nodes from the content structure appear brighter and to the right.

[1] Currently, limited Asymmetrix ToolBook integration is available, and XEmacs psgml DocBook integration is being developed.

In CHASID, concepts of the content structure represent *topic*s discussed in the document. So, this specific CHASID content structure is called a *topic map*[2]. To the user, it looks like a conceptual graph [11], but does not offer the full modelling and inference power. Operations are available to modify this graph generically by adding topics and drawing relations.

However, `rwe` has shown that constructing a sufficiently detailed content structure by way of such atomic operations requires more effort of the author than the content structure is likely to repay. CHASID thus offers *template*s, proven substructures to be instantiated en bloc in the document graph. Examples for templates are the advance organiser, the full-fledged presentation of a figure or the overall Introduction-Methods-Results-Discussion structure of an article.

Each template comes with a preview and a description of its content and usage. Within the context of a template, the nodes and relations are *weight*ed: they may be *optional*, *important* or *crucial*. The weights serve as a hint to the user, but are also used to check the consistency of the document.

When a template is instantiated, it is usually connected to the existing document graph through nodes that become parts of the template instance, too. To support this, two primary means of instantiation are offered: the fully flexible interface offers a way to name a match for every node of the template, while the so-called *pragmatic instantiation* offers only the nodes that are expected to be matched.

If a node is part of a template instance (by instantiation, match or manual addition), the weight of the node in this context is recorded. For each template instance, the *analysis* part of CHASID checks whether the instance still has all the components it requires: If crucial or important components are removed, a warning message is issued, with the tone of the message depending on the weight of the missing component. Removal of an optional component triggers no user-visible action.

Other analyses are done on relationships known to CHASID: If a topic has parts to it, then each of the parts should be discussed if the topic as a whole is to be discussed. If it is not, a warning is attached to the topic, the missing part and the division where that part of the topic should be addressed.

Such "addressing" is recorded in import and export relationships between divisions and topics. A division is said to *export* a topic if reading this division contributes substantially to understanding this topic. Conversely, a division is said to *import* a topic if at least a working knowledge of the topic is required to understand the division.

3.1 Implementation

Figure 3 gives an overview of the sub-graphs in a CHASID document graph. Each rectangle represents a sub-graph with an identifiable purpose. The core sub-graphs are framed with a thicker line. Arrows indicate distribution of knowledge

[2] There are some similarities with XML topic maps [12], but we were unaware of that activity when choosing this name.

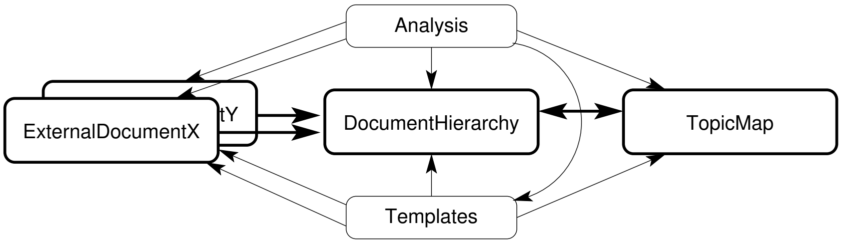


Fig. 3. Overview of the CHASID Document Model

about other sub-graphs. For example, the analysis subgraph may be removed without affecting the document hierarchy, but removing the topic map would affect the analysis.

On the left side, the *external documents* subgraphs contain reproductions of the conventional documents, specific to each conventional application. The purpose of these sub-graphs is to allow integration code with the conventional application to concentrate on communication and change propagation, so they abstract very little from the document model provided by the conventional application.

That abstraction is provided through the document hierarchy sub-graph. It consists of simple ordered trees of division nodes.

The topic map sub-graph contains nodes for the topics and relations, with edges connecting them. Types of topics are differentiated in the type system, but with only little consequences.

The analysis sub-graph contains the warning nodes generated by the analysis patterns. They may be attached to any other sub-graph, as any node may be of relevance to a detected problem.

The template sub-graph contains traces of template instances. As templates may include all types of nodes (excluding template and analysis nodes), this sub-graph is connected to all other sub-graphs, except analysis.

The modifications of this entire CHASID document graph are implemented as a programmed graph rewriting system written in PROGRES [10]. From this so-called *specification*, a C library is generated that contains functions for each operation defined in the specification. Applications based on this code are then developed by plugging the generated code into a framework that allows interactive execution of operations. Figure 4 shows a simplified overview of UPGRADE [5], the framework used here. It is also being developed at our department and used by several other projects [6, 7].

The framework offers a configurable user interface ("*UI*") with menus and buttons to invoke operations and determine parameters for them. The operations may also be invoked programmatically through an interface from Python *scripts* ("Scripts"). The C library generated from the specification ("Generated Code") then uses the PROGRES graph code library ("PGCLibrary") as its runtime environment. This in turn stores its state in the `gras` graph database ("gras"). Changes in the database cause change events to be sent to the UP-

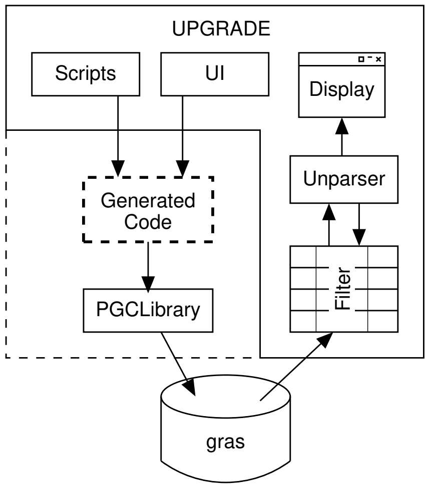


Fig. 4. Overview of the UPGRADE framework

GRADE *filter stack* ("Filter"). The lowest *filter* chiefly communicates with the data base, while further filters perform various filtering and transformation tasks. For example, an `EdgeVisibilityFilter` filters out events pertaining to edges of certain types, while an `EdgeNodeEdgeFilter` translates triples of two edges and an intermediate node to one attributed edge. The top-level filters are finally used to drive *unparser*s ("Unparser") that produce the visualisation, consuming change events and querying the filters for more information from the database. The framework defines table, tree and graph *display*s ("Display"). These are widely configurable without programming effort. Several displays may exist at the same time, each with its own filter stack.

Framework issues CHASID uses this framework for the execution of the code generated from the specification. The specifics of the implementation of CHASID are distributed across the following places:

Document model PROGRES graph operations are used to implement the document manipulations requested by the user (through the UI or document modifications in the conventional application), as well as the analysis checks. Layout information dependant on specific graph patterns is also defined here. These calculations profit most from the expressive power of graph rewriting systems.

Filters We use the filter stack parameterisation of the UPGRADE framework to create simplified views on the graph for display. Parameterised filters also hide complexity from Integration Unparsers.

Intrusion Plugins These add-ons to or modifications of the conventional application watch for changes in the conventional document, generating change notifications in the one direction, and modify the conventional document according to change requests received from integration unparsers in the other direction. They communicate with the extension's core over proprietary TCP protocols and are implemented in whatever language is most conveniently connected to the conventional application.

Unparsers Specific *display unparser*s within the UPGRADE framework control the display of subgraphs by inserting layout information into the graph view (figure 2). *Integration unparser*s interpret update events and emit corresponding change requests to the intrusion plugins.

Parsers Integration unparsers have *parser* counterparts that in turn interpret change notifications from intrusion plugins and invoke document graph operations. Integration parsers and unparsers are together responsible for spotting and breaking echo loops. Such loops may occur if the intrusion plugin detects a change in the conventional document, notifies the parser, which modifies the document graph, triggering update events, causing an integration unparser to send change requests to the intrusion plugin, which modifies the conventional document, which is detected by the intrusion plugin, and so forth.

Scripts Templates are instantiated by executing scripts, allowing the user to extend his library of templates at runtime.

Specific Modules Import of XML documents conforming to a specific DTD is implemented in additional modules calling operations of the generated code, thus playing a role comparable to the script module. Export modules to several file formats query top-level filters.

3.2 Results

The step from the purely imperative programming environment used for `rwe` to a mixed one with a large declarative graph rewriting part yielded noticeable benefits. Especially analysis checks are much more concisely described declaratively. Figure 5 shows an operation that inserts a warning node for all document hierarchy divisions that have only one son.

An equivalent imperative implementation would have to define an iteration over all division nodes, check three context conditions (has one child, no further children, no messages for this condition attached already, which includes an iteration over all messages at the division), before it can create the new message. In a reasonable layout, this amounts to more than twenty lines of code.

This difference increases as the complexity of the conditions grows. While the graph operations do get denser and harder to read, the imperative implementation scales noticeably worse and becomes unmaintainable fast.

Apart from complex pattern matches, derived attributes and automatically calculated relations are useful features missing from imperative languages. The syntax-driven PROGRES editor is a further benefit, as it checks the context-sensitive syntax at editing time and allows graphical programming.

By separating the logic (specified as graph operations) from its presentation (specified through parameterised displays) we profit from developments of the flexible and reusable UPGRADE framework.

The runtime environment for this prototype includes the proprietary graph database `gras`, the PGCLibrary, various java libraries and the java runtime.

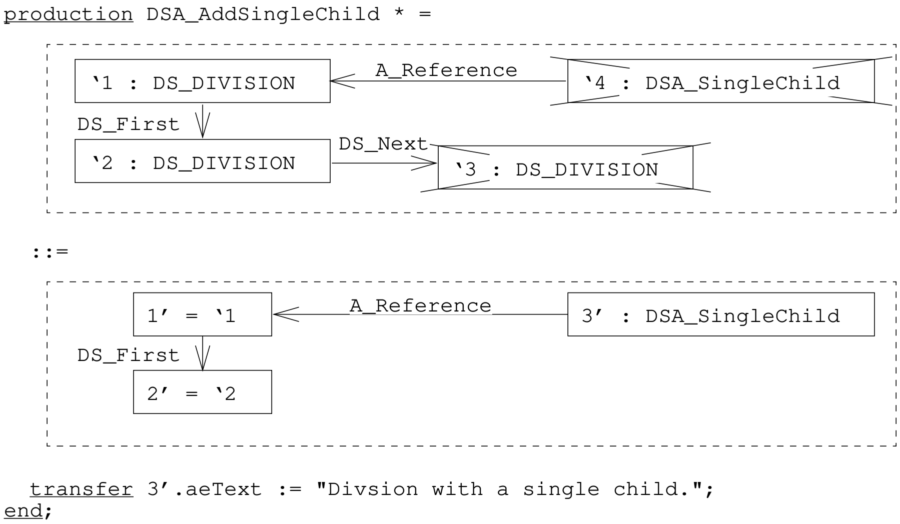


Fig. 5. Sample operation from the specification

These prerequisites restrict the prototype to specific Unix systems and cause high hardware requirements. Faults in any supporting product often impact CHASID. This clearly marks CHASID as a scientific prototype. Projects are underway to improve performance and to allow usage of more generally available object bases.

4 Comparison

`rwe` as well as CHASID allow the author to model the content structure of documents. The expressive power of a graph model is needed to describe such models. We have used both imperative and declarative ways to handle these graphs, allowing us to compare results and implementations.

4.1 Results in Structured Document Processing

While `rwe` can be used to model the structure of documents a posteriori, it is not useful for authoring new documents. CHASID with its integration of conventional applications uses the modelled content structure to also support the author in creating a consistent document. Both systems have mainly been used to experiments with various ways to build content structures and offer functionality based on it. As an experimentation field, graph-based application development has proven useful.

4.2 Developing graph-based applications

However, some specific issues are to be solved when applying graph technology in the development of medium-sized interactive applications.

Type safety is a desirable property of any larger piece of source code. It arises here as the problem of maintaining type safety across implementation languages. It has been provided in `rwe` through the use of generated encapsulation classes that enforce schema correspondence with the database. In CHASID, the specification shares the schema with the database. The PROGRES editor's analysis ensures local type safety here.

Yet in CHASID there are parts outside the specification that depend on specific graph structures and thus the schema. The filter parameters use schema identifiers (strings) to define views on the database. Unparsers filter change events for specific schema identifiers to create layout information and inform the intrusion plugins about model changes. Parsers transform the change events from intrusion plugins into generic operation calls. All these dependencies are not type-checked. There are also no generated encapsulation classes. This is a shortcoming of the new system design that necessitates expensive testing to find errors at runtime that were automatically found by the compiler in `rwe`.

A database-based application like `rwe` may be expected to cope with **schema evolution** gracefully. This has been achieved for `rwe` in that the author was able to extend the schema between runs and could keep his document graphs. If loss of documents is acceptable, further kinds of modifications were possible, unless they affected the well-known schema. This was checked at startup-time. CHASID, however, allows no modifications any more, as PROGRES needs to know the schema at editing time. If we were to allow schema extensions in CHASID we would have to store, implement and check our own type system. The specification would then contain chiefly meta-types. Types would be checked at runtime only. We prefer the safety of static checks, but other projects [6] at our department do implement dynamic type systems.

The **user interface** of `rwe` was completely under the control of `rwe`. This gives freedom, but is prohibitively expensive. In CHASID, we leave the presentation of the media content to an conventional application and use the generic display capabilities of UPGRADE to display the graph structure. This allows us to concentrate on the content structure and experiment with different models easily specified in PROGRES.

4.3 Conclusion and plans

Our applications serve as research prototypes and are sufficiently usable for that purpose. Most kinds of operations are specified graphically quicker than implemented imperatively. The stronger type safety enforced by the specification environment makes generic operations harder to achieve. Even when using a flexible framework, providing an user interface usable with only little training still requires a lot of effort.

In the near future, Emacs integration, template libraries and user interface issues are our main concerns. The content structure parts of the specification have reached a state where the remainder of the prototype needs to catch up for some real-world tests.

References

1. R. Baumann. *Ein Datenbankmanagementsystem für verteilte, integrierte Software-Entwicklungsumgebungen (A Database management system for distributed, integrated software development environments).* PhD thesis, RWTH Aachen, Wissenschaftsverlag Mainz, Aachen, 1999.
2. A. Behle, F. Gatzemeier, and O. Meyer. Graph technology for structured documents. In *Proceedings of SEKE '99*, pages 52–56. Knowledge Systems Institute, June 1999. URL http://www-i3/research/publications/by_year/1999/SEKE99.pdf.
3. F. Gatzemeier. Patterns, Schemata, and Types — Author Support through Formalized Experience. In B. Ganter and G. W. Mineau, editors, *Proc. International Conference on Conceptual Structures 2000*, volume 1867 of *LNAI*, pages 27–40. Springer, 2000. ISBN 3-540-67859-X. URL http://www-i3.informatik.rwth-aachen.de/research/publications/by_year/2000/PatScheTy-online.pdf.
4. F. Gatzemeier and O. Meyer. Improving the Publication Chain through High-Level Authoring Support. In Nagl et al. [9]. URL http://www-i3/research/publications/by_year/1999/HighLevelAuth-online.pdf.
5. D. Jäger. Generating tools from graph-based specifications. *Information and Software Technology*, 42:129–139, 2000.
6. D. Jäger, A. Schleicher, and B. Westfechtel. AHEAD: A graph-based system for modeling and managing development processes. In Nagl et al. [9], pages 325–339.
7. A. Marburger and D. Herzberg. E-CARES research project: Understanding complex legacy telecommunication systems. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 139–147, Lisbon, Portugal, 2001. IEEE Computer Society Press.
8. M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *LNCS*. Springer, Heidelberg, 1996.
9. M. Nagl, A. Schürr, and M. Münch, editors. *Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, volume 1779 of *LNCS*, Castle Rolduc, The Netherlands, Sept. 2000. Springer.
10. A. Schürr, A. Winter, and A. Zündorf. Protoyping graphbasierter Systeme. pages 86–97. GI, Oct. 1995. GI SofTec NRW (1995).
11. J. Sowa. Conceptual graphs: Draft proposed american national standard. In W. Cyre and W. Tepfenhart, editors, *Conceptual Structures: Standards and Practices*, volume 1640 of *LNAI/LNCS*, pages 1–65, Heidelberg, 1999. Springer. ISBN 3-540-66223-5. URL http://www.bestweb.net/~sowa/cg/cgdpans.htm.
12. TopicMaps.Org Authoring Group. XML Topic Maps (XTM) 1.0. http://www.topicmaps.org/xtm/index.html, Aug. 2001.
13. S. Toulmin, R. Rieke, and A. Janik. *An introduction to reasoning.* Macmillan Publishing co., Inc., New York, first edition, 1979.

Graph Rewriting Techniques and Tools for MPEG-7 Multimedia Description Schemes

Hélène Jacquet, Hawley Rising III, and Ali Tabatabai

Sony Electronics Inc., Media Processing Division,
Network & Software Technology Center of America,
3300 Zanker Road, San Jose, California 95134, USA
{helene.jacquet,hawley.rising,ali.tabatabai}@am.sony.com

Abstract. MPEG-7 is a standards activity of the Moving Picture Expert Group which aims to provide standardized tools and methods to define ways for associating and exchanging data related to a multimedia content. Therefore, MPEG-7 is not about compression, such were MPEG-2 or MPEG-4, but rather about metadata. A central concept in MPEG-7 is the notion of description, used to organize data about the multimedia content according to their mutual relationships. This naturally leads to complex graph-based constructs for which graph-rewriting techniques are seen to be of great interest. The aim of this paper is to give an overview of the MPEG-7 standard as well as an idea of the use of graphs and graph rewriting techniques related to it. We hope in this way, to introduce MPEG-7 to the graph grammar community, and seek the involvement of researchers in the development of architectures or tools devoted to the creation, the management, the access or the interoperability of MPEG-7 descriptions.

1 Introduction

MPEG-7 is a standards activity of the Moving Picture Expert Group to provide *"a multimedia content description interface"*. It consists of a standardized set of tools and methods for construction and exchange of (object-oriented) descriptions of multimedia content features, with the aim of allowing interoperability among applications and devices for searching, indexing, filtering and access of audio-visual content. Multimedia must be understood here as any form of audio or visual (or both) document. MPEG-7 is not aimed at any particular application but rather at standardizing tools that will support as larger applications as possible. This standard offers uniform ways to describe elementary pieces of information, related to the content, linked together according to their relationship in the context of the multimedia document. That can be either (but not only) :

- specific features of a content (color, shape) linked together by their spatial relationship (east, over);
- (meta)data related to the multimedia document(such as author, information about rights management);

– radically high level information contained or suggested by the content itself: soccer players linked by the action depicted by the audio-visual segment (on-the-left, gives-the-ball-to) entraining a spectators' reaction (happiness, angriness) detected from the audio.

Such uniformity needs a high level of abstraction when dealing either with the definition of the basic building blocks or when establishing the useful tools to construct descriptions. Due to their ability in providing an expressive, flexible and visual (and also popular) data representation, graphs can be seen as an omnipresent underlying structure in several part of the MPEG-7 standard. On the second author's initiative, the graph-rewriting paradigm was proposed as a useful tool for the construction and the manipulation of those graph-based descriptions. It has been accepted and is part of the normative elements called *Graphical Term Definition* and *Graphical Classification Scheme* (see [1]).

The next section gives a brief presentation of the MPEG-7 standard with an emphasis on part 5, the *Multimedia Description Scheme (MDS)*, which defines the set of tools needed to construct graph-based descriptions. The third section specifically discusses the part of the standard in which graphs grammars have been introduced. The fourth section gives a formal presentation of the *MPEG-7 graphs* as well as a brief presentation of the current graph rewriting methods which are part of the standard. Finally, in the conclusion, we discuss the reasons why we believe that the keys to the use of graph rewriting mechanism in relation with future MPEG-7 based applications, are, in part, "in the hands" of the graph rewriting community.

2 An overview of the MPEG-7 Standard

Experts, in MPEG-7, come from disparate subjects with separate backgrounds (databases, signal processing, knowledge management, digital libraries), and from diverse circles with specific needs (electronics companies, universities, content owners, web based application providers etc.). As a matter of fact, features of multimedia content can be understood as low level features (color, shape), information about the content (creation date, creator) as well as semantics (what represents the content) but also about the access to the content (user preferences). In other words, MPEG-7 tools must reconcile all the shapes that a description of multimedia content can take, and be adapted to many kinds of multimedia applications. In addition, to satisfy those primary goals, the main challenges for this new content description format are to ensure a minimal common understanding that is crucial for interoperability, and to provide efficient solutions for storage and fast transport. The standard MPEG-7 is divided into 8 parts : the complete architecture of the standard is described in [12]. For the rest of this paper, we will essentially focus on the *description tools* and particularly on the *Multimedia Description Scheme*(part 5).

2.1 MPEG-7 normative elements

The standard specifies four types of normative elements divided into two kind of tools : the first kind is related to the *description tools* (and covers part 2 to 5 of MPEG-7 activities, and the second is related to *the systems tools* (part 1 of MPEG-7 activities) whose aims to support delivery of descriptions, multiplexing of descriptions with multimedia content, synchronization, file format and so on. The *description tools* specify how to construct complex (graph-based) descriptions using the following fundamental blocks :

– **The Descriptors (D)** are the basic units to describe features, attributes, or groups of attributes of multimedia content. They are designed primarily to describe audio-visual features (color, texture) or attributes of the content (location, time, quality).
– **The Description Schemes (DS)** describe entities or relationships pertaining to multimedia content. The DSs are graph-based descriptions used to define higher-level features related to the content and produce more complex descriptions of the content (regions, segments, objects, permanent metadata related to the creation, rights, etc.). Components of **DS** may be Descriptors, or datatypes as well as DS themselves.
– **The Datatypes** are the basic reusable datatypes employed by Description Schemes and Descriptors.

Finally, the Description Definition Language (DDL) (defined in the part 2 of the MPEG-7 standard) specifies the syntax of the Ds and the DSs: to insure a wide interoperability, DDL is a variant of W3C XML Schema Language.

2.2 Multimedia Description Schemes (MDS)

The MPEG-7 MDS set of tools provides a way to associate to a multimedia document a triple *Description-Package-Metadata* (see also [13]). The *Description* is a structured set of information about the content, descriptions can be either complete or partial. A complete description provides a standalone description of the content for an application. A partial description, also called a *Description Unit*, is an independent piece of description dedicated to part of the content, or to a “view” of the content. A partial description can be added to a complete description, or it can be used by an application as partial information about the content. The *Package* contains information about the internal organization of the description, allowing, for example, specific searches into the description. *Metadata* contains information related to the description such that date of the creation of the description, usage of the description, confidence on the description and so on. A *Description* is a graph-based representation that starts with a *root element*. The root element essentially indicates if the description is complete or partial. If the description is complete, the root element is followed by a *top-level* element that gives information about the task of the description. Basically, a description is related either to the content itself or to the management of the content and can describe the nature of the content (video, images, etc.) as well as the semantics of the content (i.e. what represents the content) or information on the usage of the content.

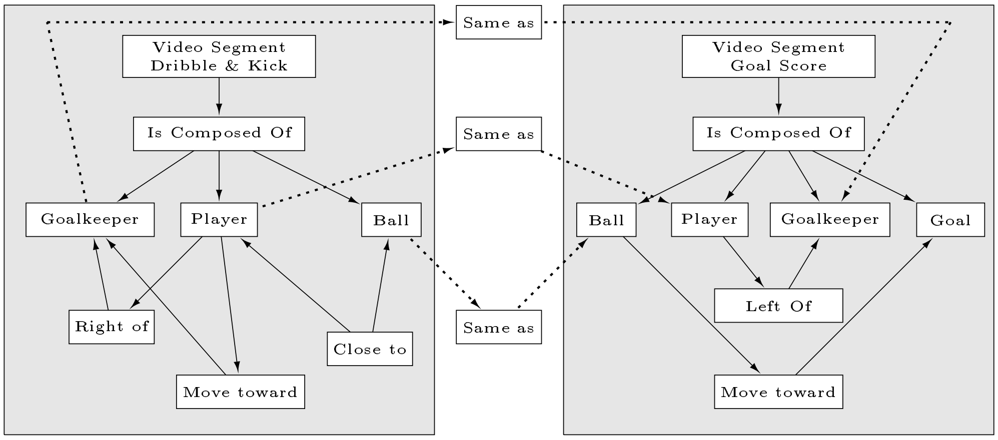


Fig. 1. A structural description of two successive video-segments.

Figure 1 shows a possible graph-based description of two video-segments called *Dribble and Kick* and *Goal Score*: it describes the structure of the content with three "moving regions" (identified by the terms *Ball, Player, Goalkeeper*) and the spatial relationship between them (*Right of, Move toward*), then the same three "moving regions" plus a "still region" (*Goal*) in the next action. In the left most segment, the player is moving toward the goalkeeper while staying on his left: the ball and the player are staying close together. In the second segment, the player is now on the right of the goalkeeper and the ball is moving toward the goal. The dashed edges establish the correspondence between the moving regions on the two video-segments. Here, this description is not related to the semantics of the content, but is merely a structural description. Actually, the real elements of the description are "moving regions" (that is physical shapes that can be detected into the content) and relations between them are mainly physical. One expects that some parts of such a structural description may be automatically extracted from the content it-self.

3 Introduction of graph rewriting in MPEG-7 standard

In the previous example, we said that the description is merely structural, and suggested that the words are just used to denote regions of the content without involving any semantics. In other circumstances, and because the current state of the technology is far away to be able to infer meaning from only low level descriptors, one may wish to introduce higher level of information about the semantics of the content. MPEG-7 provides several tools to introduce semantic information in a description. This ranges from simple key-words added as attributes to a segment in order to add meaning, to the possibility of constructing complex descriptions related to the semantics suggested by the content. In order to guarantee a minimum agreement on the meaning, and to be able to exchange such descriptions, MPEG-7 proposes to control the vocabulary used in any part of a description, by means of *Terms* and *Classification Scheme*.

3.1 Terms and Classification Schemes

MPEG-7 provides a set of "normative terms" (essentially those that are used to define the MPEG-7 format itself), as well as a set of tools to define vocabulary that can be used in the descriptions. Those tools are :

- A format allowing the definition of new *Terms* : each *Term* possesses a name, a unique identifier, a definition setting its meaning, and, possibly a set of relations with others *Terms*. A *term* is always defined in reference to a context or a domain represented by a *Classification Scheme*.
- A *Classification Scheme* is a hierarchical organization of *Terms* related to a particular domain. This can be seen as an ontological representation, or a thesaurus (in reference to knowledge representation and language processing fields respectively). The default relation between a term and it subordinates is "narrower in meaning than" but other kinds of relations can be used as well as user-defined ones.

Figure 2 shows (a piece of) the definition of a *Classification Scheme* establishing the possible spatial relations between segments: it's extracted from the MPEG-7 standard and expressed in the XML-based DDL language. This *Classification Scheme* defines several terms that should be used in descriptions : here, *Binary, Directional, Left, Right, Topological* and *Equal* are defined.

```
<ClassificationScheme uri="urn:mpeg:mpeg7:cs:SpatialSegmentRelationCS">
 <Definition> Spatial relations among segments </Definition>
       <Term termID="binary">
       <Definition> Spatial relations between two segments </Definition>
           <Term termID="directional">
           <Definition>
            Spatial relations that describe how two segments are placed and relate
            to each other in space
           </Definition> [...]
                <Term termID="left">
                <Definition> If segment B is left of segment C, B.x.e <= C.x.s </Definition>
                </Term>
                <Term termID="right">
                <Definition> Inverse relation of left </Definition>
                </Term>[..]
            </Term>
            <Term termID="topological">
            <Definition> [...]
                 <Term termID="equal">
                 <Definition>If segment B equals segment C, B = C </Definition>
                 </Term>[..]
             </Term>
         </Term>
</ClassificationScheme>
```

Fig. 2. Part (11.8.2.1) of MDS - Definition of SpatialSegmentRelation CS.

An implicit relation between those terms is also induced by the tree structure of the definition's syntax : for example, the term *Left* is a specialization of *Directional* and *Directional* itself, is a specialization of *Binary*.

3.2 From Classification Scheme to Graphical Classification Scheme

In the definition of *SpatialSegmentRelation* (cf Figure.2), each term of the classification scheme inherits of the definition of its ancestor. In addition it possesses its own definition that gives it a meaning and that constrains its usage. To allow more complex ways of defining and adding constraints on the use of terms in accordance with the Classification Scheme it refers to, *MDS* introduces the notion of *Graphical Classification Scheme*. In a *Graphical Classification Scheme*, a term is defined by a *Graphical Term Definition* that is none other than a graph structure together with rules setting its use and its possible ways of being replaced.

Let us take the example of the term *"Left-of"* defined in the previous classification scheme. According to the definition given by the scheme on Figure 2, this term is used to qualify a spatial relation between two segments that carries an information about their mutual direction and that verifies a particular property. It means therefore that it can be used in a description involving two segments B and C to replace a relation *Directional* if the attributes of the two segments B and C verify the property $B.x.e \leq C.x.s$. As B and C are certainly part of a bigger description, this change would certainly have some consequences on the whole description. For example, the transitivity of the *Left-of* relation would induce that each segment on the left of B would then be on the $left$ of C. In a similar manner, as *Left-of* is the inverse relation of *Right-of*, it also means that each segment that was on the right of C is now on the right of B and so on.

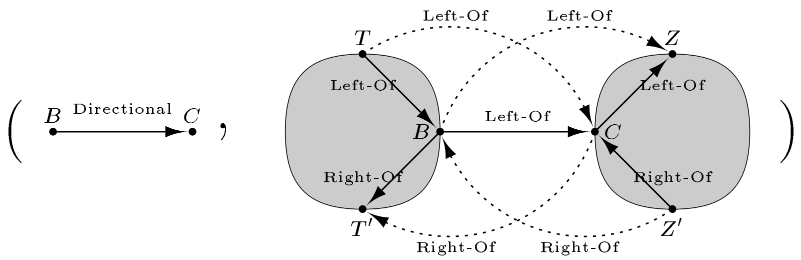


Fig. 3. Replacing *Directional* by *Left-of* with a graph rewriting rule

This may be translated in graph formalism by representing the description with a graph where nodes are complex attributed objects standing for segments and edges are relations (binary, directed here) between them. For this example, we are only concerned with labels on the edges whose are terms used to qualify the relation. The replacement of the *Directional* relation by *Left-of* may be expressed by a graph production with application conditions (on the attributes of the objects), and an embedding mechanism which specifies ways to create new edges in the derived graph. Such a production is represented in an informal way on Figure 3 : the grey part of the production represents the neighborhood of the nodes B and C in the mother graph while the dotted edges stand to describe the connection relation after the rewriting step. It means that each node T (respectively T') that was linked in the mother graph to B by a *Left-of* (respectively *Right-of*) edge would be linked to C by a *Left-of* (respectively *Right-of*) edge after the rewriting step. The same kind of embedding is described for the neighbors of C. The result is that, in a description conforming to the *Graphical Classification Scheme* where the term *Left-of* is associated to this *Graphical Definition*, one would be allowed to replace a *Directional* relation by a *Left-of* relation according to the production established in Figure 3.

As MPEG-7 standardizes descriptors for low level feature of a multimedia content but do not aim to standardize how they must be used for, CBIR (Content Based Image Retrieval) for example, *MDS* here introduces via the graphical term definition, the ability to use graph rewriting for the manipulation of complex descriptions, without imposing any obligation or ways to use them. With a little bit of imagination, we can see that there are situations where the ability of rewriting graph-based descriptions may be helpful: refining descriptions as the above example, gluing descriptions together, controlling or validating the structure of a description, restructuring a description, simplifying a description, going from the *Dribble and Kick* video segment description to the *Goal Score* description, and so on.

3.3 About the motivation for using graph rewriting in MPEG-7

Graph grammars have been specifically introduced into MPEG-7 with objective to cope with high level semantic description of content[1] and are indirectly inspired by works in relation with algebraic semiotics. Without entering into the details, semiotics is an important contribution to the understanding of cognitive process for knowledge, communication and exchange influenced by the Pierce's sign theory. This approach is gaining popularity in several fields of computer sciences and, particularly, when dealing with user interaction concerns. We refer the reader to [6] for a detailed information.

Generally speaking, a sign can be seen as "whatever drives basic information to construct meaning". Three kinds of signs are commonly admitted to exist: *icons*, that convey ideas of the things they represent, then *indications, or indices'* which show something about things, and *symbols*, or general signs, which have become associated with their meanings by usage.

[1] We have already mentioned that MPEG-7 propose tools to describe content according to many aspects : semantic meaning of the content is one of them.

In [7], Goguen et al., establish a category for sign systems suitable to capture the following important ideas :

- Signs appears as members of signs systems, not an isolation.
- Most signs are complex objects, constructed from other lower level signs.
- Signs systems are better viewed as theories - that is, declarations for symbols plus sentences, called "axioms", that restrict their use - than as (set-based) models.
- Representations in general are *morphisms* (mappings) between signs systems.

In [6], G. Fauconnier et al, reveals that an important mode of composition of signs system called *blending* is directly involved into the construction of superior knowledge, understanding and communication. Goguen et al. show that this operation may be mathematically defined by the operation of amalgamate sum, also called *pushout*, in the category for sign systems. ¿From then, it appeared to the people involved in the MPEG-7 effort that allowing tools for the composition of graph representations[2] may be of particular interest for the construction of elaborated descriptions of the semantic of the content. The importance of the notion of *mapping* in semiotics as well as the huge need of formalizing abstractions, explain by itself the fundamental reasons for the introduction of graph grammars methods in MPEG-7 via the use of approaches known as *algebraic*. We will briefly present those approaches in the next section.

4 Graphs and Graph Grammars in MPEG-7

More than just their ability on describing complex datastructures, graphs are used in the MPEG-7 standard for their capacity on representing *relations* (in the mathematical sense) between entities. Actually, the notion of relation is central in the set of tools developed by MPEG-7 as they allow the construction of higher level of information by putting together basic pieces of information.

To cover the wide range and variety of graph-based representation involved into the standard, a generic definition for graph have been chosen by the standard committee : MPEG-7 graphs are labeled polyadic graphs according to the definition [8].

Definition 1. *A* polyadic graph *is a system* $G = \langle V_G, E_G, s_G : E_G \to V_G^*, t_G : E_G \to V_G^+ \rangle$ *where* V_G *is a set of* nodes*,* E_G *a set of* edges*,* s_G *is a map called the* source map *and* t_G *the* target map*. By convenience, we will call* sources *of* e *the images of an edge* e *by* s_G *and* targets *of* e *its images by* t_G*.*
A labeled polyadic graph *is a triple* $\langle G, l_n, l_e \rangle$ *where* G *is a polyadic graph,* $l_n : V_G \to \mathcal{A}_n$ *is a mapping form the nodes of* G *to a set of nodes labels and* $l_e : E_G \to \mathcal{A}_e$ *is a mapping from the edges of* G *to an set of edges labels.*

[2] Graph descriptions, when used to describe the semantic of the content, may be seen as a simplified form of signs system.

It results from this definition that a polyadic graph describes a set of (m,n)-ary relations e (also called an $m+n$-ary relation) between nodes, where m is the cardinality of the set of sources of e and n the cardinality of it set of targets. In addition, those (m,n)-ary relations are ordered in the sense that there is an order for the appearance of the sources in the $s_G(e)$ sequence and, respectively, an order for the appearance of the targets in the $t_G(e)$ sequence. By convention, a unary relation is a $(0,1)$-relation, that is with an empty set of sources. For those that are more familiar with hypergraphs, a polyadic graph corresponds to a directed hypergraph with ordered tentacles (as defined, among others, in [10]). It is also obvious that a polyadic graph with $(1,1)$-relations is a binary directed graph (in the usual sense).

Figure.4 shows "a" representation of the polyadic graph G where $V_G = \{x_i, \forall i \in [1,8]\}$, $E_G = \{e\}$, $s_G(e) = (x_1x_2x_3x_4x_5, x_6x_7x_8x_5)$ that comprise one edge e and eight nodes. This polyadic graph is then composed by a unique (5,4)-relation represented by an (hyper)edge with 5 incoming tentacles and 4 outgoing tentacles. In Figure 4, the square node stands for this (hyper)edge e, the left-most nodes and the top one are images of e under the source map, whereas the rightmost nodes and the top one are images of e under the target map.

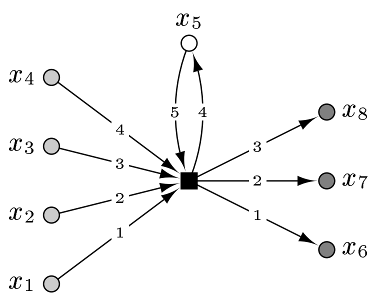


Fig. 4. Representation of a polyadic graph.

4.1 Graph rewriting approaches in MPEG-7

Motivations for the introduction of graph grammars in MPEG-7 have been presented during the previous sections. Here, we only intend to enumerate the approaches already inscribed in the standard.

The graph rewriting paradigm essentially finds justification in the MPEG-7 standard in its ability to give a compact, clear, and abstract framework to describe the dynamical behaviors of graphs. That is, no decision was made about a "right paradigm" for graph transformation for multimedia description, and several approaches were standardized in order to cover graph operations in the most general sense and allow convenience over a wide range of possible applications. The graph rewriting approaches that are now part of the MPEG-7 standard are those known under the name of *categorical* (or *algebraic*) approaches as they define graph transformation by the way of operations in the graph category (and more generally in the polyadic-graphs category). We enumerate each of them in the following and give, for each, their characteristics according to three criteria :

- Are they suitable to describe node, edge and subgraph replacement ?
- Are they defined for graphs, hypergraphs or polyadic graphs ?
- Do they involve an embedding mechanism (edge oriented) or a connecting mechanism (node oriented) ?

Pullback rewriting in the sense of [2] gives a categorical framework to describe node replacement in *structured graphs*. This approach primarily uses a pullback operation (based on the categorical product) adequately combined with a particular graph called *Alphabet Graph* to describe node rewriting in graphs. It has been extended to hypergraphs (and any kind of graph that has a particular structure) by viewing hypergraphs as the objects of a category over a structured graph. The use of the pullback approach for polyadic graphs can be found in [10] where polyadic graphs are called directed hypergraphs. This approach defines a connecting mechanism but is able to simulate an embedding mechanism.

Double Pullback rewriting is an extension of the pullback approach for node oriented sub-graph rewriting in structured graphs. This approach describes the replacement of a whole subgraph as a double step operation: first the collapsing of the sub-graph to a single meta-node by construction of a complement (that is, the inverse operation of the pullback) then, the replacement of this meta-node by a pullback rewriting step. One must also notice that the existence of the pullback complement is subject to identified application conditions. A description of this method for hypergraphs can be found in [2].

Pushout rewriting is used to describe edge rewriting in graphs (or hypergraphs). The notion of *pushout rewriting* refers to the use of the categorical pushout as an operation to glue together two graphs according to a morphism that defines the gluing points, while optionally deleting some edges or nodes. The introduction of the pushout operation in MPEG-7 has been done in reference to two different approaches.

- First, the *single pushout approach* for graph rewriting defined in [11] where a pushout method based on partial graph morphisms is used to describe hyperedge and hypergraph rewriting. This approach, as all the edge oriented approaches, involves an embedding mechanism.
- Second, as we already mentioned in the previous section, the use of pushouts to describe the semiotic operation called *"blending"*. In [7], a 3/2-category is stated to give an algebraic framework for semiotics. In this category, the pushout operation, when it exists, is used to describe a particular gluing of graphs called *blending*.

Double Pushout Rewriting (DPO) as defined in [9] and recently overviewed in [5]. In the DPO approach, hyperedge (and more generally hypergraph) replacement is described as a cut and paste operation in hypergraphs. It is done in two steps : the first step removes a part of the hypergraph by constructing an inverse pushout operation (called pushout complement) while the second step glues a new hypergraph in place of the removed one. The DPO approach involves an embedding mechanism : it must also been note that the pushout complement doesn't necessary exists, leading then to some well-known application conditions. This formalism has certainly been the most intensively studied approach over the thirty past years and has been shown as usefully applicable in several circumstances.

5 Conclusion

In the standard itself, graph rewriting paradigm is proposed as a useful tool to help the control of the structural aspects of a description. However, no precise global approach to exercise this structural control is established. The main reason is that MPEG-7 doesn't aim to give solutions regarding to any particular application or use of descriptions that could be done. As it is pointed out in [3], a modern architecture for a multimedia content management would certainly be strongly dependent (but not only) on the particular domain and application type. If multimedia description schemes are clearly designed to be a tool tending toward a better interoperability between multimedia management applications, the way developers would use this tool is, hopefully, totally open. We think that graph transformation methods can be helpful to deal with some fundamental problems suggested MPEG-7 :

- Providing visual, and generic tools to construct, validate and maintain schema-based MPEG-7 compliant annotations.
- Proposing concepts or methods to insure the interoperability between different MPEG-7 based applications, but also with already existing applications or databases.
- Inventing new architectures for applications that would exploit the whole informative aspects captured by this new emerging standard.

It is obvious that the graph rewriting community could find interest in playing a role in further development related to MPEG-7 standard. It may lead to the improvement of recent tools and theories and to the demonstration of the usefulness of this paradigm. In addition, the application is in a domain, the multimedia industry, that is expanding. We see two possible issues for the involvement of the graph rewriting community : first, supporting those who might be interested in developing applications MPEG-7 compliant and that would be inclined to use graph transformation. This can be done either by providing accessible information on the "success stories" and about the available tools, or by the proposition of inventive solutions. We fully agree with remarks done in [4] on the difficulties, for a new comer to graph rewriting, on understanding its usefulness. In addition, when dealing with multimedia applications, urgency to provide the market with new tools needs fast answers on the potential adequacy of graph transformation methods to accomplishing precise goals. Second, proposing changes (through the Appligraph project?) directly to new versions, or evolution of the standard. Actually, while the first version of the standard is fixed, there is a process open to further adjustments and ideas.

MPEG-7 home page : `http://www.darmstadt.gmd.de/mobile/MPEG7/`

References

1. Text of ISO/IEC FDIS 15938-5 Information Technology: Multimedia Content Description Interface, Part 5. Multimedia Description Schemes, N4242 (2001) System
2. Bauderon, M. Jacquet, H. : Pullback as Generic graph rewriting mechanism. Applied Categorical Structures, Volume. 9, No. 1, Kluwer Academic Publishers, (January 2001) 65 – 82
3. Behrendt, W., Fiddian, N. J., Gray, W.A.: Database interoperation support in multimedia applications : architecture and methodology. Electronics & communication engineering journal, Volume. 13. Issue. 4 , (Aug. 2001) 173–182
4. Blostein, D., Fahmy, H., Grbavec, A. : Issues in the Practical Use of Graph Rewriting, In Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science; LNCS 1073, Springer (1996) 38–55
5. Corradini, A., Montanari, U., Rossi, F. : Algebraic Approaches to Graph Transformation : Basic Concepts and Double Pushout Approach, in Handbook of Graph Grammars and Computing by Graph Transformation, G. Rozenberg Edt, Volume. 1, World Scientific (1996) 163–240
6. Fauconnier, G. : Mapping in Thought and Language, Cambridge University Press, (1997), Book
7. Goguen, J.A. : An introduction to Algebraic Semiotics, with Application to User Interface Design, in Computation for Metaphor, Analogy and Agents, edited by Chrystopher Nehaniv, Springer Lecture Notes in Artificial Intelligence, Volume. 1562, (1999) 242–291
8. Goguen, J.A., Burstall, R.M. : Some Fundamental Algebraic Tools for the Semantics of Computation, Part 1: Comma Categories, Colimits, Signatures and Theories. Theoretical Computer Science, Vol. 31 (1984) 175–209
9. Ehrig, H : Introduction to the algebraic theory of graph grammars, in Graph Grammars and their applications to Computer Science, LNCS No. 73, (1979) 1–69
10. Jacquet, H., Klempien-Hinrichs, R. : Node Replacement in Hypergraphs : Translating NCE Rewriting : In procedings of TAGT'98, Theory and Application of Graph Transformations, LNCS No. 1764, Springer (1999) 117 – 130
11. Löwe, M. : Algebraic approach to Single Pushout graph transformation, TCS, Volume. 109, (1993) 181 – 224
12. Martinez, José M.: Overview of the MPEG-7 Standard (version 5). ISO/IEC JTC1/SC29/WG11 N4031 (March 2001)
13. Salembier, P., Smith, J.R.: MPEG-7 multimedia description schemes. Circuits and Systems for Video Technology, IEEE Transactions on circuits and systems for video technology, Vol. 11 Issue. 6 , (June 2001) 748 –759

Constructing Shapely Nested Graph Transformations*

Frank Drewes[1], Berthold Hoffmann[2], and Mark Minas[3]

[1] Umeå Universitet, drewes@cs.umu.se
[2] Universität Bremen, hof@informatik.uni-bremen.de
[3] Universität Erlangen-Nürnberg, minas@informatik.uni-erlangen.de

Abstract. *Shapely nested graph transformation* is the computational model for DiaPlan, a language for programming with graphs representing diagrams that is currently being developed. This model supports nested structuring of graphs, graph variables, and structural graph types (*shapes*), but is still intuitive. In this paper, we show that the *construction* of shapely nested graph transformation steps can be reduced to solving the subgraph isomorphism and variable matching problem for the components of a structured graph, and devise restrictions of the transformation rules that improve efficiency. Shapes provide useful structural information about the graphs involved in a transformation step, and may therefore further improve efficiency.

1 Introduction

DiaGen [19] is a tool for implementing the *syntax* of diagram languages. The editors generated by DiaGen represent diagrams as graphs, perform scanning and structure editing by graph transformation, and parse their syntax according to a graph grammar. DiaPlan [13, 15], a programming language that is currently being designed by the authors, shall complement DiaGen by a tool for implementing the *semantics* of diagrams.

Shapely nested graph transformation [14] has been devised as the computational model of DiaPlan. This model is rather intuitive, although it supports powerful concepts that are not found in other graph transformation languages like Progres [23] and Agg [9]:

- Edges may contain graphs in a nested fashion, for a compositional *structuring* of graphs.
- *Graph variables* allow subgraphs of arbitrary size to be duplicated, compared, or deleted in a single transformation step.
- The admissible *shape* of graphs can be specified by syntactic rules that allow for type checking.

* The second author has been partially supported by the ESPRIT Working Group *Applications of Graph Transformation* (Appligraph).

In general, graph transformation is difficult to implement. So this paper studies the *construction* of shapely nested graph transformation steps. We show that this construction can be reduced to subgraph isomorphism and variable matching for the components of a graph, and propose restrictions of the transformation rules that improve efficiency, in particular by cutting down nondeterminism.

We investigate nested graph transformation (without shapes) in Section 2. Shapes are introduced afterwards, in Section 3, because shapely nested graph transformation leads to considerably different algorithms. Related and future work is discussed in Section 4.

2 Nested Graph Transformation

Our notion of graphs follows [14]; it is more general than usual ones: edges may connect an arbitrary number of nodes, not just two, and they may contain graphs in a nested fashion. We also distinguish a sequence of interface nodes at which graphs may be glued together.

More precisely, let L be a *ranked alphabet* where every symbol $l \in L$ comes with an *arity* $\mathrm{arity}(l) \geqslant 0$. The set $\mathcal{G}$ of *graphs* over L consists of sixtuples

$$G = \langle V, E, \mathrm{lab}\colon E \to L, \mathrm{att}\colon E \to V^*, \mathrm{cts}\colon E \to \mathcal{G}, p \in V^* \rangle^1$$

with finite sets V of *nodes* and E of *edges*, where every edge $e \in E$ has a *labelling* $\mathrm{lab}(e)$, a sequence $\mathrm{att}(e)$ of $\mathrm{arity}(\mathrm{lab}(e))$ *attached nodes*, and a *contents* $\mathrm{cts}(e)$, and where p designates a sequence of *points*.[2] It is required that p does not contain repetitions, and that the same holds for every $\mathrm{att}(e)$ ($e \in E$). This is a well-known normal form which does not restrict the expressiveness of the concepts defined below.

By $\langle\rangle$ we denote the *empty graph*; the *handle graph* $\langle l \rangle$ of a label l consists of an edge e with $\mathrm{cts}(e) = \langle\rangle$ that is labelled with l and attached to $\mathrm{arity}(l)$ points. In a graph G, an edge e is called a *frame* if $\mathrm{cts}(e) \neq \langle\rangle$, and *plain* otherwise; e is qualified as a *k-ary l-edge* if it has k attachments and label l. G is called *plain* if it contains no frames, and k-ary if it has k points.

The tree-like nesting of frames in a graph G as above defines *nested edges*

$$\Delta_G = \{\varepsilon\} \cup \{ew \mid e \in E, w \in \Delta_{\mathrm{cts}(e)}\}$$

for selecting the *subcomponent* G/w contained in a nested edge $w \in \Delta_G$, and *assigning* a graph U to that nested edge, written $G[w \leftarrow U]$. By $G(w)$ we denote the *plain graph at w*: the one which is obtained from G/w by replacing the contents of each frame with $\langle\rangle$.

Two graphs G and H are *isomorphic*, written $G \cong H$ if they are equal up to the identities of their nodes and edges.

---

[1] V^* denotes the set of *sequences* over some vocabulary V. The *empty sequence* is denoted by ε.

[2] A more precise definition would define graphs by induction over the nesting depth of edges.

In contrast to notions of *hierarchical graphs* that are used for system modeling [8], our nesting concept is *compositional*: it forbids edges between components so that component assignment is possible. This is important for programming.

Figure 2 below shows three *control flow graphs*. Their nodes represent execution states, and their edges represent assignments, branches, and procedure calls; calls are frames that contain the control flow graph of the called procedure.

Variables. Let X be a ranked alphabet of *variable names* disjoint with L. A graph P over $L \cup X$ is called a *pattern* if all its *nested variables* (the nested edges labelled by X) are plain. By $\underline{P}$ we denote the *skeleton* of a pattern P where all variables have been removed.

Let P be a pattern with a k-ary nested variable $we \in \Delta_P$. The *replacement* of e in P/w by a k-ary graph U is then defined by gluing the attachments of e to the corresponding points of U, removing e, and assigning the result to P/w.

A pattern C with a single nested variable we is called a *context*. The *embedding* of a graph U in C is denoted as $C[U]$ and defined by replacing the nested variable we by U.

A function $\sigma\colon X \to \mathcal{G}$ is a *substitution* if it maps variable names onto graphs with the same arity.

The *instantiation* of a pattern P according to σ is obtained by the simultaneous replacement of all nested variables we in P by the graph $\sigma(\mathrm{lab}_{G/w}(e))$; the resulting *instance graph* is denoted by $P\sigma$.

Graph Transformation. Using the notions summarized above, transformation rules and steps can be defined in a similar way as in the area of term rewriting [16]. In doing so, it seems sensible to apply the same restrictions as for the rules of a term rewrite system: Their left-hand side patterns must not be variables, as such rules apply to every graph so that the system diverges, and their right-hand side patterns must not contain uninstantiated variables, since then arbitrary substructures have to be created “out of thin air”.

A *(transformation) rule* $P \to_t R$ consists of two patterns P and R such that the left-hand side P is not a variable handle, and only variable names from P occur in the right-hand side R. Then t *transforms* a graph G into another graph H, written $G \Rightarrow_t H$, if t can be instantiated with a substitution σ, and embedded into some context C so that $G \cong C[P\sigma]$ and $H \cong C[R\sigma]$.

Example 3 (Control Flow Graph Transformation). In Figure 1 we show a rule l that performs a *loop transformation*, and a rule u that *unfolds* the body of a procedure call. Figure 2 shows two transformations of a control flow graph using l and u. The context for the first step equals the graph $G[f \leftarrow \langle D\rangle]$ (where f is the frame in G, and $\langle D\rangle$ is D's handle graph), and the substitution maps D onto the handle graph $\langle x := e\rangle$. For the second step, the context equals the graph obtained by replacing the frame f by a D-variable, and the substitution maps D onto H/f.

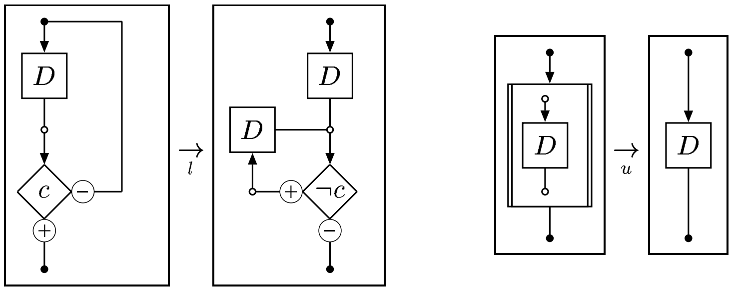


Fig. 1. Rules for loop transformation and for procedure unfold

Let l^{-1} and u^{-1} be obtained by interchanging the sides in rules l and u. Then u^{-1} is not a rule because its left-hand side is the handle graph $\langle D \rangle$. Indeed this rule could always be applied, to *fold* any control flow graph to a procedure call. However, l^{-1} is a transformation rule that could transform H back to G.

Note that a single graph transformation may affect arbitrary large subgraphs of the host graph. Every application of l *duplicates* the subgraph bound to the variable name D. Similarly, a rule *deletes* the subgraph bound to a variable name in its left-hand side if that name does not occur in its right-hand side. And, a rule may require to *compare* arbitrarily large subgraphs: the rule l^{-1} applies only to a host graph like H, where both D-variables on its left-hand side match isomorphic subgraphs. This is a rather complex applicability condition, and therefore often forbidden in applications based on term rewriting. So implementations of (shapely) nested graph transformation may also require that rules are left-linear, i.e. that every variable name occurs at most once on the left-hand side.

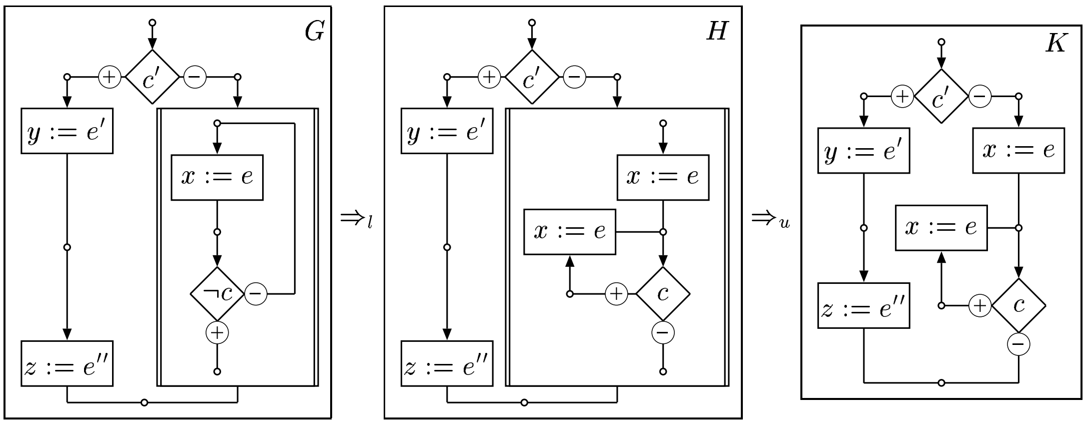


Fig. 2. Two transformation steps

Construction of Transformations. For a graph G and a rule $P \to_t R$, every valid transformation step can be constructed as follows:

1. FIND P's skeleton $\underline{P}$ in G, i.e. determine an *occurrence graph* O such that $\underline{P} \cong O \subseteq G/w$ for some nested frame w in G.
2. BIND P's variable names by a *matching substitution* σ such that $P\sigma \cong G/w$.[3]
3. REWRITE, i.e. determine H as $G[w \leftarrow R\sigma]$.

For plain graphs, FIND and BIND together constitute the *graph matching problem* studied in [20]. Obviously FIND cannot be solved without solving the *subgraph isomorphism problem*, which is NP-complete. BIND is somewhat easier: Using the techniques presented in [20] it is solvable in polynomial time. Once this has been done, REWRITE is easy.

A closer look at FIND and BIND reveals that the main difficulty is to match the plain graph $P(u)$ against $G(wv)$, for nested frames u and wv. From such local solutions the required global one can be constructed rather efficiently. Thus, nesting alone makes FIND and BIND more efficient if graphs consist of many small components, since these can be considered in isolation. (Notice the benefit of compositionality.)

However, BIND may still produce an exponential number of matching substitutions for some occurrence. If evaluation is done with backtracking (as in PROLOG), all of them may have to be tried out, eventually. It is thus important to cut down the number of matches. Fortunately, rules may be restricted so that they have at most one matching substitution for any occurrence:

Theorem 1. *Let $\underline{P} \cong O \subseteq G/w$, where no variable in $P(\varepsilon)$ is attached to a point. The* BIND *step can yield at most one matching substitution σ so that $P\sigma \cong G/w$, provided that every $u \in \Delta_P$ satisfies one of the following properties:*

1. *The variables in $P(u)$ have pairwise disjoint sets of attached nodes and $G(wu)$ is connected,*
2. *$P(u)$ contains at most one variable,*
3. *$P(u)$ is a handle graph or contains no variable at all.*

The conditions 1–3 are ordered with increasing strength. Rule u in Figure 1 of Example 3 satisfies condition 3. So do all the rules used for specifying a graphical version of Quicksort in [5]. (Actually also the right-hand sides of all rules in that paper satisfy condition 3.) This indicates that even rigidly restricted variable concepts suffice for many nontrivial programming situations.

Rule l^{-1} in Figure 1 of Example 3 violates condition 2 since the variables of its left-hand side share one of their attached nodes. Rule l in that figure, although satisfying condition 2, fails as the D-variable on its left-hand side is attached to

[3] To be precise, one must extend P by a *hole variable* h that is attached to P's points. Then $G[w \leftarrow \sigma(\mathrm{lab}_P(h))]$ determines the skeleton of the context C for the transformation. Furthermore, substitution σ has to be consistent with the occurrence of $\underline{P}$ in G which has been found in the FIND step, i.e., the occurrence morphism has to be a submorphism of the isomorphism $P\sigma \to G/w$.

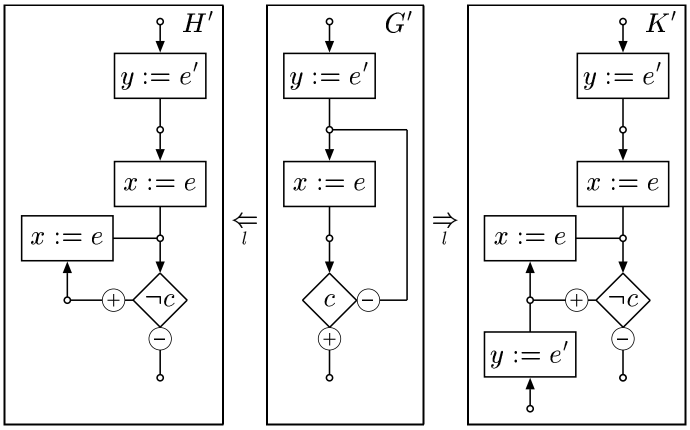


Fig. 3. Two loop transformations

a point. Figure 3 illustrates that the occurrence of l's pattern skeleton may then be transformed with different instantiations. Our intuition about control flow graphs may mislead us to believe that l transforms G' in just one way, yielding H'. In that transformation, the subgraph $\langle y := e' \rangle$ in G' is part of the context wherein the instance of l is embedded. However, this subgraph may also belong to the instantiation of l. Then, the transformation yields the graph K', where the subgraph $\langle y := e' \rangle$ is duplicated, but introduced as dead code that will never be executed.

3 Shaped Nested Graph Transformation

Figure 3 points out an inherent feature of nested graph transformation: All graphs and pattern over the label alphabets may occur as host graphs, in substitutions of variables, and in rules. The construction of transformations has to cope with the general case, even if the graphs that actually occur have particular properties. (For example, all control flow graphs are connected, and have a unique start node.) These properties could be used to construct transformations more efficiently.

Therefore we devise rules specifying the syntax of graphs in a context-free way so that graphs and patterns can be checked against these rules.

Syntax Graphs. Let N be a ranked alphabet of *nonterminals* disjoint with the vocabularies L and X. Graphs over $L \cup N$ are called *syntax graphs* if their N-edges are plain (as for patterns).

Let Σ be a finite set of *syntactic rules* of the form $n ::= R$, where n is a nonterminal, and R is a syntax graph with arity(n) points. A *direct derivation* of a syntax graph G to a syntax graph H under Σ, written $G \Rightarrow_\Sigma H$, is obtained

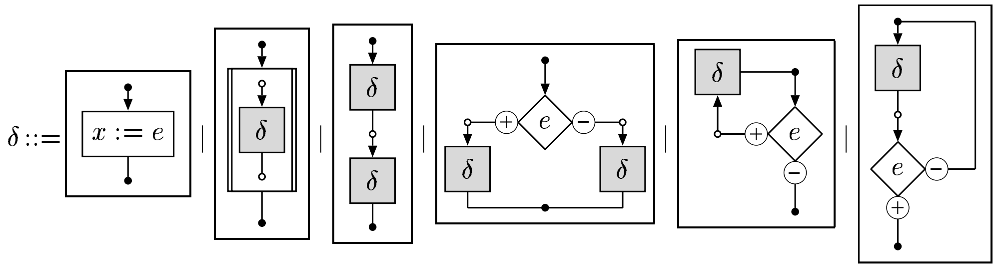


Fig. 4. A grammar for structured control flow graphs

by replacing a nested n-edge we in G by R, where $n ::= R$ is a syntactic rule in Σ. The reflexive and transitive closure of $\Rightarrow_\Sigma$ is denoted by $\Rightarrow_\Sigma^*$.

Example 4 (Grammar for Structured Control Flow Graphs). Figure 4 shows the rules defining the syntax of *structured control flow graphs* that are built (from left to right) over assignment and procedure call by sequential composition, conditional statement, pre-checked and post-checked loop. In this example, δ is the only nonterminal. We write $\delta ::= R_1 | \cdots | R_n$ to abbreviate several rules $\delta ::= R_1, \ldots, \delta ::= R_n$ for the same nonterminal.

Syntactic rules specify *hyperedge replacement*, which derives one of the best-studied classes of context-free graph languages (see [11, 4] for details). Such rules allow to define recursive "algebraic" data types of functional and logical languages, and sophisticated pointer structures like cyclic lists, or leaf-connected trees, which cannot be defined in imperative languages (see also [10]).

Membership in these languages is decidable:

Theorem 2. *The question "$\langle n \rangle \Rightarrow_\Sigma^* G$?" is decidable for all syntax graphs G.*

Shaped Graphs and Patterns. For the rest of this paper, we fix a finite set Σ of syntactic rules over the nonterminals N, and use it to specify shapes of graphs. (The term *shape analysis* is used for inferring properties of pointer structures in imperative programs [22].) We assume that every variable name $x \in X$ is *typed* with a nonterminal $\mathrm{type}(x) \in N$.

The *shape* $\lceil P \rfloor$ of a pattern P is the syntax graph obtained by relabelling every x-variable in P by its type.

A pattern P is *shaped* by some nonterminal $n \in N$ if $\langle n \rangle \Rightarrow_\Sigma^* \lceil P \rfloor$ (or just *shaped* if n is not relevant). A shaped context C is called *n-context* if its unique variable is of type $n \in N$. (Note that in general, C may be shaped by another nonterminal.) A substitution σ is *shaped* if the graph $\sigma(x)$ is shaped by $\mathrm{type}(x)$ for all $x \in X$.

Shapely Transformation. We now refine graph transformation so that it preserves shapes.

A rule $P \to_t R$ is *shapely* if P and R are shaped by the same nonterminal, say n. Then t *transforms* a graph G into another graph H, written $G \Rrightarrow_t H$, if t can be instantiated with a shaped substitution σ, and embedded into some n-context C so that $G \cong C[P\sigma]$ and $H \cong C[R\sigma]$.

Our definition of shapes is consistent, since the result of a shapely transformation is a shaped graph again (see [14] for the straightforward proof).

Theorem 3. *In the situation above, G and H are of the same shape as C.*

Altogether, shapes set up a *type discipline* that can be *statically checked*: Theorem 2 allows to confirm whether a set T of transformation rules is shapely or not. If the rules are shapely, and a graph G (the "input") has been checked to be shaped, Theorem 3 guarantees that every transformation sequence $G \Rightarrow^* H$ will yield a shaped "output" graph H. Type-checking between the steps ("at runtime") is not necessary.

Example 5 (Shapely Control Flow Graph Transformations). The patterns in Figure 1 of Example 3 are shaped according to δ, so the rules l, U, and l^{-1} are shapely. The contexts of the transformations in Figure 2 are δ-contexts, and the matching substitutions are shaped so that the transformations are shapely as well.

In Figure 3, the transformation $G' \Rightarrow_{l^{-1}} H'$ is shapely. However, the transformation $G' \Rightarrow_l K'$ is not shapely, because neither the context, nor the substitution used in it are shaped.

Construction of Shapely Transformations. As pointed out earlier, matching a pattern P against (a subcomponent of) G can be done by computing matches of the plain patterns $P(u)$ against the plain graphs $G(v)$. The obtained results can be combined using a top-down or bottom-up procedure. Let us briefly describe the bottom-up case; the top-down procedure is similar. Assume we are given a matching algorithm for plain graphs. In the first step we consider all $u \in \Delta_P$ such that P/u is plain. For every $v \in \Delta_G$ we use the given algorithm to determine whether $P(u) = P/u$ matches $G(v) = G/v$. Next, we consider all $u \in \Delta_P$ such that P/u has nesting depth 1 and repeat the procedure, applying the given algorithm to $P(u)$ and each $G(v)$ to find out whether P/u matches G/v (making use of the already computed information). This is repeated until we reach the root of P.

Obviously, the recursive part of this procedure can be implemented efficiently. It can be used to find a single matching, but also to enumerate all possibilities if an evaluation strategy involving backtracking is desired or needed.

Thus, the complexity is mainly determined by the complexity of matching $P(u)$ against $G(v)$. As mentioned above, this problem generalizes the subgraph-isomorphism problem and is thus not efficiently solvable unless P=NP. However, in the presence of shapes one does not need to solve the problem in all its generality. Below, we briefly discuss some possibilities to gain efficiency.

Functional languages correspond to the (very restrictive) case where the $G(v)$ are taken from a finite set. More precisely, a term $f(t_1, \ldots, t_k)$ is represented by a graph consisting of an f-labelled frame containing a graph with k points and k frames $e_1, \ldots, e_k$. Frame e_i is attached to the ith point and represents, recursively, the subterm t_i. In this case, plain graph matching can be done in constant time and thus occurrences can be found efficiently.

In more general cases, one can make use of algorithms which compute the set of all derivation trees of $G(v)$ (with respect to the shape grammar; see [4] for a discussion of derivation trees). If this can be done in polynomial time, matchings can usually be found in polynomial time as well. This is because P is also shaped, and hence each $P(u)$ can be represented by its set of derivation trees. The latter can be matched against the derivation trees of $G(v)$, which essentially reduces the problem to the question of tree matching.

Unfortunately, parsing of hyperedge-replacement languages is NP-complete in general [17], so this approach is not always useful. However, restrictions under which parsing becomes polynomial have been studied by Lautemann, Vogler, and Drewes [18, 24, 3]. Let us discuss the way in which the algorithm by Lautemann can be used. The algorithm is a generalization of the well-known parsing algorithm by Cocke, Kasami, and Younger [25]. It can be reformulated in such a way that it returns a representation of all derivation trees of the (plain) input graph. Since there may be exponentially many derivation trees, sharing is used to represent them on polynomial space. In other words, the returned representation of the forest of derivation trees is a directed acyclic graph (dag).

To explain the structure of this *derivation dag* let us first discuss a possible representation of a derivation tree of an n-shaped plain graph $H = G(v)$. Let $n ::= R$ be the rule applied to $\langle n \rangle$ in the first step of the derivation, where R contains k nonterminal edges $e_1, \ldots, e_k$. Hence, H is obtained from an isomorphic copy R' of R by replacing each e_i with a graph $H_i \subseteq H$ which is derivable from $\langle \mathrm{lab}_R(e_i) \rangle$. To account for this fact, the derivation tree has a root node v that represents the pair (n, H) and a frame with contents R' which is attached to $v_1, \ldots, v_k, v$ where $v_1, \ldots, v_k$ are the root nodes of the derivation trees obtained recursively from the derivations $\langle \mathrm{lab}_R(e_i) \rangle \Rightarrow_{\Sigma}^{*} H_i$. Now, the derivation dag D representing the set of all derivation trees of H is obtained by taking the forest of all these derivation trees and identifying all nodes which represent the same pair (n', H').[4] Under the conditions of [18] the algorithm by Lautemann computes D in polynomial time. In particular, the size of D is polynomial. The mentioned conditions basically state that the graphs generated by the shape grammar fall apart into at most a constant number of components by deleting k nodes, where k is the maximum type of nonterminal edges. The shape grammar for control flow graphs discussed in Example 4 satisfies this requirement.

[4] Here, "the same" really means that the graphs are identical, not just isomorphic, since we are working with concrete subgraphs of H.

As mentioned above, the considered component $P(u)$ of the pattern graph can be turned into a corresponding derivation dag D' as well, since the allowed patterns are shaped with respect to the same shape grammar. Then it is not hard to find all possible matchings by a bottom-up or top-down procedure that runs in time $\mathrm{O}(d \cdot d')$, where d and d' are the sizes of D and D'. In fact, it should be mentioned that this performs even the BIND step discussed in Section 2 since each matching found in this way associates every variable in $P(u)$ with a node in D. The graph represented by this node is the one to be bound to the variable.

Even though it runs in polynomial time, from the point of view of efficiency the procedure just described has the drawback that the derivation dag D' must be reconstructed after each step. This is because the application of a rule may invalidate some of the derivation trees represented in D while on the other hand creating new possible derivation trees. However, the specific derivation tree to which D' is mapped is, by the shapedness of rules, turned into a correct derivation tree of the resulting graph by performing the corresponding replacement on the level of trees. If we have unique derivation trees, we can therefore represent graphs by their derivation trees throughout and reduce transformation to the (much more efficient) replacement of subtrees of derivation trees. As such, this option is not very realistic because the restriction to grammars with unique derivation trees would be much to strong. However, unlike the class of grammars considered by Lautemann, those studied in [24, 3] have unique derivation trees modulo a certain type of associativity and commutativity rules. We do not wish to go into the details here, but it seems quite clear that the technique sketched above can be extended in order to work in this, somewhat more realistic case as well. For instance, the syntax rules in Example 4 have unique derivation trees, modulo the third rule that expresses associativity of sequencing in control flow graphs.

It is worth pointing out that implementations may select, for each $G(v)$, the matching algorithm which is most suitable for its shape. Hence, a mix of different matching algorithms may be applied to match P against G. For example, if Σ contains shapes n and m which satisfy the restrictions of matching algorithms A respectively B we may use algorithm A for n-shaped components whereas B is used for m-shaped components. In this way, efficient algorithms can be applied whenever possible without restricting the language in general.

4 Conclusions

Nested graph transformation is closely related to other ways of graph transformation. On the one hand, it lifts the substitutive transformation of flat graphs [20] to nested graphs; on the other hand, it extends hierarchical graph transformation [5] with respect to the use of variables. Hierarchical graph transformation has in turn been defined by lifting double pushout graph transformation [6] to hierarchical graphs (for injective occurrences, as studied in [12]). The paper [8] defines double pushout transformation of hierarchical graphs where edges may cross the border of components (called *packages*), yet without investigating under which conditions the hierarchy stays intact. A general framework for the transformation of (many kinds of) hierarchical graphs is developed in [2]. Shape specifications (for plain graphs) have been considered in Structured Gamma [10].

This paper indicates that nested graph transformation is not only intuitive and expressive, but may also be implemented in a reasonable way. Nesting helps for the general case, and Theorem 1 gives reasonable conditions that eliminate the overhead for binding. Shapes may even improve these results since the conditions on the pattern graphs may be relaxed if the host graphs have structural properties that simplify the task of finding matchings and bindings. The investigation of such structural properties is an important and interesting question for future work.

Several other questions remain to be studied as well. Let us focus our discussion on shapes here. The syntactic rules can be extended by *embedding rules* [19], without sacrificing Theorem 2. Then one can also specify non-context-free shapes like, for instance, general control-flow diagrams (of "spaghetti programs"). The parsing algorithm explained in [1] allows more general syntax rules. However, it is unclear how efficient it will be in practice, and more important, whether it is consistent with the operations of context embedding and variable instantiation that are fundamental with our way of graph transformation.

Acknowledgment We thank the referees for useful comments.

References

1. P. Bottoni, A. Schürr, and G. Taentzer. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In M. M. Burnett et al., editors, *Proc. VL'2000*. IEEE Press, 2000. Full version appeared as Technical Report SI-2000-06 at the University of Rome.
2. G. Busatto. *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation*. Dissertation, Universität Paderborn, 2002. Forthcoming.
3. F. Drewes. Recognising k–connected hypergraphs in cubic time. *Theoretical Computer Science*, 109:83–122, 1993.
4. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [21], chapter 2, pages 95–162.
5. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, to appear 2002. (A short version appeared in number 1784 of Lecture Notes in Computer Science, pages 98–113, 2000).
6. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 1–69. Springer, 1979.
7. G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Specification and Programming*. World Scientific, Singapore, 1999.
8. G. Engels and R. Heckel. Graph transformation as a conceptual and formal framework for system modelling and evolution. In U. Montanari, J. Rolim, and E. Welz, editors, *Automata, Languages, and Programming (ICALP 2000 Proc.)*, number 1853 in Lecture Notes in Computer Science, pages 127–150. Springer, 2000.

9. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In Engels et al. [7], chapter 14, pages 551–603.
10. P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2/3):263–289, 1998.
11. A. Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.
12. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
13. B. Hoffmann. From graph transformation to rule-based programming with diagrams. In M. Nagl, A. Schürr, and M. Münch, editors, *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE *'99), Selected Papers*, number 1779 in Lecture Notes in Computer Science, pages 165–180. Springer, 2000.
14. B. Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.
15. B. Hoffmann and M. Minas. Towards rule-based visual programming of generic visual systems. In N. Dershowitz and C. Kirchner, editors, *Proc. Workshop on Rule-Based Languages*, Montréal, Quebeq, Canada, Sept. 2000.
16. J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.
17. K.-J. Lange and E. Welzl. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16:17–30, 1987.
18. C. Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421, 1990.
19. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 2002. To appear.
20. D. Plump and A. Habel. Graph unification and matching. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
21. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
22. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.
23. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Engels et al. [7], chapter 13, pages 487–550.
24. W. Vogler. Recognizing edge replacement graph languages in cubic time. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Fourth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 532 of *Lecture Notes in Computer Science*, pages 676–687. Springer, 1991.
25. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208, 1967.

Modeling the Pickup-and-Delivery Problem with Structured Graph Transformation*
– A Case Study –

Renate Klempien-Hinrichs, Peter Knirsch, Sabine Kuske

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

Abstract. The paper presents a first case study modeling the Pickup-and-Delivery Problem (PDP) with structured graph transformation. It discusses the logistic background of the PDP and describes the structuring concept of transformation units which is employed to model the PDP. The problem domain of the PDP is visualized by a graph representing a map with a set of jobs. The distribution of the jobs to transportation means is performed via the controlled application of graph transformation rules embodied in transformation units. In a first step the PDP with maximal load and time windows is modeled. Secondly, the model is extended to include a limited number of vehicles starting from and returning to their home depots.

1 Introduction

Traditionally, logistics is concerned on a more or less abstract level with transporting entities. The problem becomes the more concrete the more constraints have to be satisfied: e.g. maximal capacity of transports, time windows, maximal number of transporters, depots of transporters, and so on. In the literature, several variants of the problem are studied and algorithmic solutions discussed (see e.g. [SS95]). The algorithms must be intelligible, for which visualization is helpful. This holds especially if some heuristics is employed, such as scheduling of transports by hand.

In every transport problem, an underlying structure can be found which is some type of map. Since Euler's solution to the Königsberg bridge problem [Eul36], it is known that a map can be seen as a graph, with nodes representing locations and edges the links between pairs of locations. This suggests modeling transport problems and developing algorithms in the framework of rule-based graph transformation.

Graph transformation is a well-developed field [Roz97,EEKR99,EKMR99] and has many application domains, such as database systems, abstract data

* This work was partially supported by the ESPRIT Working Group Applications of Graph Transformation (APPLIGRAPH).

types, theorem proving, functional programming languages, visual languages, distributed systems, the unified modeling language UML, etc. The basic idea of graph transformation is to represent system states as graphs and system evolution steps as applications of graph transformation rules. In the literature there exist various approaches to graph transformation. A detailed description of the major ones can be found in [Roz97].

For the modelization of dynamic systems with graph transformation, large sets of graph transformation rules may be involved and have to be managed in a meaningful way. For this reason various structuring concepts for graph transformation systems have been developed (cf. [HEET99]) which allow to modularize large graph transformation systems. One of these concepts is that of transformation units [KK99,Kus00], which has the advantage of being independent of the underlying graph transformation approach.

This paper is a first case study modeling the Pickup-and-Delivery Problem with transformation units. In Section 2, the logistic background of the problem is detailed. In Section 3, transformation units are recalled. In Section 4, the pickup-and-delivery problem with maximal load and time windows is modeled by graph transformation, and in Section 5 this model is extended to include a limited number of vehicles starting from and returning to their home depots. The paper concludes with Section 6.

2 Logistic Background

In the modern business landscape, which is characterized by highly competitive, transparent, global markets and over-capacities, logistics as an integrating discipline is viewed as one of the main potentials to identify and save costs and thereby gain competitive advantages. Its scope lies on all processes from the procuring of raw materials to delivering the final product and beyond. The overall complexity of logistic scenarios can be imagined by having a look at a somehow naive definition of logistics called the *Seven Rs*: Logistics is concerned with ensuring availability of the right product, in the right quantity and the right condition, at the right place, at the right time, for the right customer, at the right cost. Such a multi-dimensional problem is of course hard to deal with, even if we restrict ourselves to smaller subproblems or just compute approximations.

In our case study we concentrate on the transportation domain, an important field in logistics research. The objective of the Pickup-and-Delivery Problem (PDP) is to find a route that enables one to pick up and deliver goods at predetermined locations. Shipment centres or courier services are possible examples. The problem is comparable to the Traveling Salesperson Problem, where all locations have to be visited on a route. The PDP has the additional restriction that here pickup places have to precede delivery places. From the applied point of view such problems were first investigated in [Wil70]. A survey of algorithms for the class of PDP problems can be found e.g. in [SS95].

Adding to the most rudimentary version of the PDP the constraints of maximal vehicle load and time windows (more side constraints are considered in

Table 1. Specification of the jobs

id	src	dst	wt	edt	lat
1	A	D	5	7	16
2	A	C	12	8	14
3	B	C	3	10	17

Table 2. Distances between locations

	A	B	C	D
A	0	3	4	∞
B	3	0	2	∞
C	4	2	0	3
D	∞	∞	3	0

Section 5), we have the following characteristics of the problem: There is an unlimited fleet of vehicles, with the capacity of each vehicle restricted to *maxwt*. Furthermore, there is a given finite set (*JOBS*) of *jobs* that have to be processed. Each job is specified by its *identification* (*id*), its *weight* (*wt*), a *source* (*src*), and a *destination* (*dst*). Moreover, the *earliest departure time* (*edt*) and the *latest arrival time* (*lat*) determine a time window in which the transport has to take place. We assume that loads can be split arbitrarily and transshipment is allowed, i.e. during a transport some of the freight might change the vehicle.

A sample set of jobs is given in Table 1, which could be an excerpt from a relational database system. Table 2 states the distances (typically measured as average travel time) between the locations; the entry ∞ indicates the absence of a direct connection. Furthermore, let the maximal load of a vehicle be 10 (tons).

In order to find a solution for this PDP instance, visualizing the topology of the locations is helpful. A graph represents this information in a natural way, as Figure 1 illustrates: The nodes stand for the different places, the edges show which place can be reached directly from a specific place, and the edge labels store the time it takes to get there. A graph of this kind will be called a *map*. A visualization of the jobs is integrated in the next section.

An algorithm solving the PDP with maximal vehicle load and time windows must find a feasible schedule of the jobs so that the time and load constraints are obeyed. Additional suitable aims are to minimize the number of vehicles, or the total distance any of the vehicles has to cover. Although less tangible, one may also try to maximize customer satisfaction, which is hard to express in figures.

The algorithm presented in Sections 4 differs from the ordinary ones in particular because we make use of nondeterminism to overcome the complexity. The results are always feasible schedules, but most of them are not optimal according to the above-mentioned criteria, i.e. the minimality of used resources. Still, they may serve as a starting point for further optimization, as input for the *k-opt* algorithm (cf. [LK73]) or as first generation for a *genetic algorithm* as described in [Mic96].

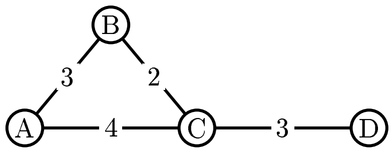


Fig. 1. Locations and distances represented in a map

3 Transformation Units

Transformation units are a modularization concept for graph transformation systems. A graph transformation system consists mainly of a set of graph transformation rules, which are applied to graphs in order to transform them. In the following we give an informal presentation of transformation units. For formal definitions the reader is referred to [KK99,Kus00].

The class of graph transformation rules used in the present case study is based on the so-called double pushout approach to graph transformation [CEH$^+$97]. A *graph transformation rule* consists of a left- and a right-hand side graph, where some subgraph is common to both these graphs. Figure 2 shows an example of a graph transformation rule.

As we will see below, this rule allows to insert the job with source *src*, destination *dst*, identification *id*, weight *wt*, earliest departure time *edt*, and latest arrival time *lat* into a map such as the one depicted in Figure 1. The left-hand side (on the left of the arrow) of the rule *insert-job* consists of two nodes; one of them is labeled with the source of the job and the other one with the destination. The right-hand side (on the right of the arrow) shows a graph that represents the job with identification *id*. The additional *ok?*-labeled node is used later to verify feasibility constraints. The common part of the left- and the right-hand side is equal to the left-hand side in this case.

The fundamental steps of *applying* a rule to a graph G are the following:

1. Choose an (identical) image of the left-hand side in G.
2. Remove the image from G up to the image of the common part.
3. Insert the right-hand side into G such that the common part is mapped to its image.

The rule *insert-job* allows to insert a graphical representation of a job into a map. Three applications of this rule to the map of Figure 1, with the variables *id*, *src*, *dst*, *wt*, *edt*, and *lat* instantiated according to the respective values from Table 1, yields the map with jobs as shown in Figure 3.

A *transformation unit* consists of an initial graph class expression, a set of imported transformation units, a set of rules, a control condition, and a terminal graph class expression. The initial graph class expression specifies all graphs a transformation may start with, and the terminal graph class expression specifies all graphs a transformation may end up with. The set of rules contains all rules

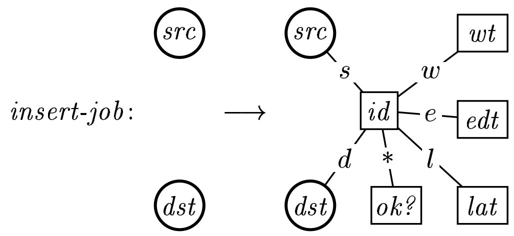


Fig. 2. The graph transformation rule *insert-job*

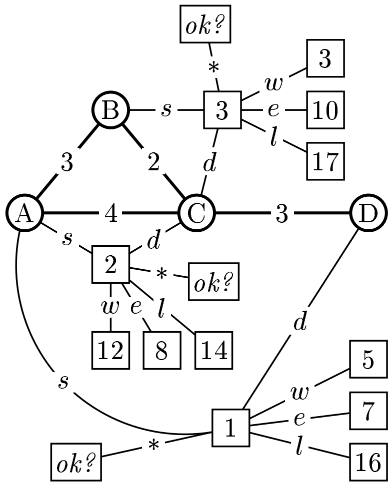


Fig. 3. The map from Figure 1 with the jobs from Table 1

which may be applied by the transformation unit. The imported units may also be applied in a transformation. The control condition regulates the graph transformation process, i.e. it serves to cut down the nondeterminism inherent to rule-based systems. Figure 4 shows two examples of transformation units.

The transformation unit *insert-job*$(id, wt, src, dst, edt, lat)$ contains the rule *insert-job* from Figure 2. The control condition requires that the rule be applied exactly once. The unit contains no explicit graph class expressions. This means that all graphs are allowed as initial or terminal graphs. Moreover, the unit *insert-job* does not import other transformation units.

The transformation unit *schedule*$(JOBS)$ accepts all maps as initial graphs, i.e. all graphs the nodes of which represent places and the edges distances between the places. It imports the unit *insert-job*, and a transformation unit with name *compute* which is developed in the next section. The unit *schedule* contains no rule, and the control condition calls the unit *insert-job* once for every job in the given set *JOBS*. Afterwards *schedule* applies the transformation unit *compute*. The terminal graph class expression *no ok?* requires that the terminal graphs do not have nodes which are labeled with *ok?*.

insert-job$(id, wt, src, dst, edt, lat)$

rules: *insert-job*

conds: *insert-job*

schedule$(JOBS)$

initial: map

uses: *insert-job*, *compute*

conds: for each $id \in JOBS$:
insert-job$(i, wt_i, src_i, dst_i, edt_i, lat_i)$;
compute

terminal: no *ok?*

Fig. 4. The transformation units *insert-job* and *schedule*

Semantically, a transformation unit transforms initial graphs into terminal graphs by applying local rules and imported transformation units such that the control condition is satisfied. More precisely, the *semantics* of a transformation unit is a binary relation on graphs containing a pair (G, G') if

- G is specified by the initial graph class expression and G' by the terminal graph class expression;
- there is a sequence $G_0, \ldots, G_n$ $(n \geq 0)$ such that $G_0 = G$, $G_n = G'$, and for $i = 1, \ldots n$, G_i is obtained from G_{i-1} by applying a rule or an imported transformation unit;
- the pair (G, G') is allowed by the control condition.

The semantics of the transformation unit *insert-job* consists of all pairs (G, G') of graphs where G' is obtained from G by applying the rule *insert-job* exactly once. The transformation unit *schedule* specifies all pairs (G, G') of graphs where G is a map, G' does not contain *ok?*-labeled nodes, and G' is obtained from G by first inserting every job of *JOBS* into G and then applying the unit *compute*. Assuming that *ok?*-labeled nodes are only removed if the jobs they refer to satisfy the feasibility constraints, the unit *schedule* may be seen as a specification of an instance of the PDP with capacity and time windows.

As already demonstrated in the examples, transformation units can be provided with formal parameters, leading to the notion of parameterized transformation units [Kus01]. Roughly speaking, a parameterized transformation unit contains a set of typed variables which may be substituted by expressions of the same type. If the expressions are variable-free, the result is a non-parameterized unit. The transformation units of Figure 4 are parameterized with the formal parameters *src*, *dst*, *JOBS*, etc.

4 Modeling the Pickup-and-Delivery Problem

Before studying our algorithm to solve the PDP with capacity and time windows, let us briefly summarize from the preceding sections the specification of a PDP instance as a graph.

A *map* is a simple undirected graph where the nodes have mutually distinct labels representing locations (e.g. by their names or post-codes), and edges are labeled with natural numbers representing distances (e.g. in traveling time). In pictures, maps are drawn with bold lines and round nodes.

A *job* is represented by four rectangular nodes: Three are labeled with natural numbers for the weight of the freight, the earliest departure time, and the latest arrival time, respectively; moreover, they are connected to the first node – labeled with the identity of the job – by edges labeled w, e, and l, respectively. In addition, a job has s- and d-labeled edges between the identity node and the source and destination locations, respectively. Finally, an edge from the identity node to an *ok?*-labeled node indicates that the job must be checked for feasibility.

As explained in Section 2, a job is feasible if (a) its weight does not exceed the maximal load *maxwt* of the transporting vehicles, and (b) its time window

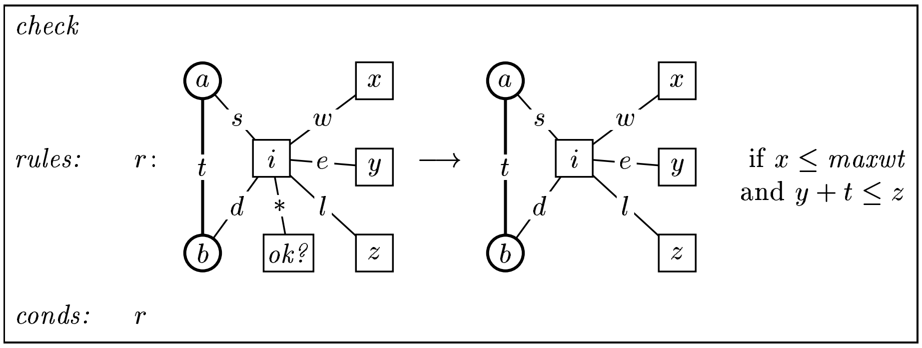


Fig. 5. The transformation unit *check*

is large enough to allow transportation along a known route. These conditions are verified by the transformation unit *check* shown in Figure 5, which finds a job in a graph and removes the *ok?*-labeled neighbor of the job identity node if the job meets the constraints.

As we assume that loads may be split arbitrarily and transshipment is allowed, a job violating feasibility condition (a) can simply be divided into two (or recursively more) subjobs, with the same source, destination, earliest departure and latest arrival time as the original job. The corresponding rule constitutes transformation unit *split-freight* shown in Figure 6. It could be applied to job 2 in Figure 3, splitting it in two jobs of weight e.g. 10 (i.e. *maxwt*) and 2, respectively.

A violation of feasibility condition (b) can have either of two reasons: no direct connection between the source and destination of the job, or a too narrow time window for the distance to cover. Performing first a shortest-path algorithm on the map (see, e.g., [DKKK00]) before starting any computation would avoid the first difficulty, and allow to detect the second. However, adding new connections between locations would obscure the actual path along which a job is taken. So, we propose a dual solution, i.e. we allow to split a job from location a to location c by introducing an intermediary stop in location b as long as the latest arrival time for the first subjob coincides with the earliest departure time of

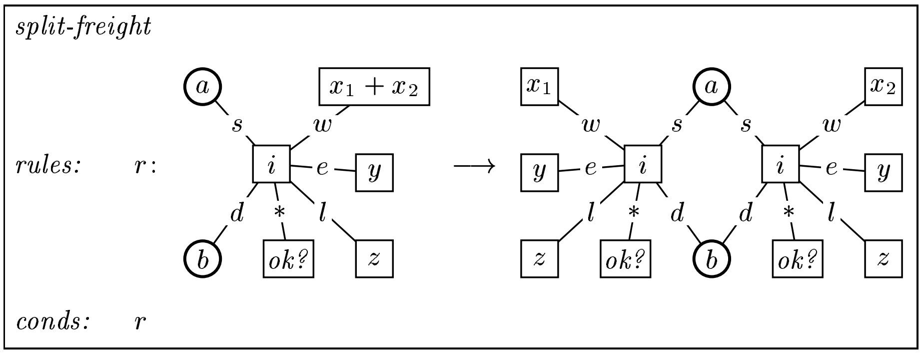


Fig. 6. The transformation unit *split-freight*

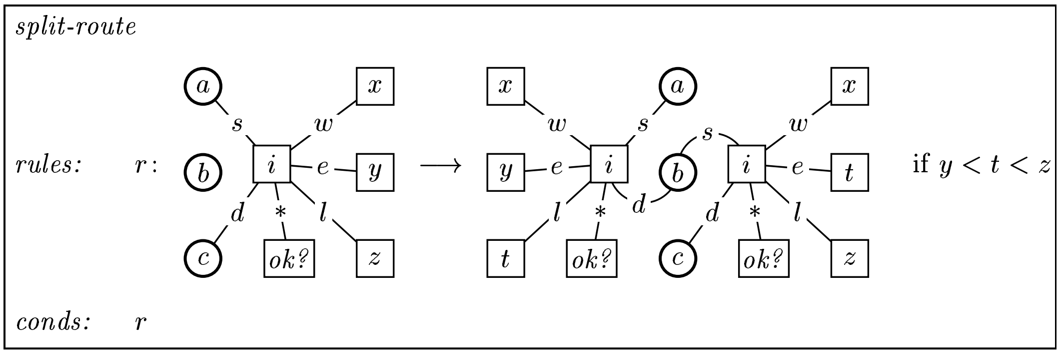


Fig. 7. The transformation unit *split-route*

the second and lies in between the earliest departure and the latest arrival time of the original job. This is implemented in the transformation unit *split-route* shown in Figure 7.

The possibility to add an intermediate stop to the route a job takes has a further advantage. Consider jobs 1 and 3 in Figure 3. If job 1 were to be carried from A to D via B and C, job 3 could be taken care of en route. Transporting at the same time two jobs with the same source and destination is expressed by merging the jobs and computing the new weight, earliest departure and latest arrival time as shown in the transformation unit *merge-jobs* in Figure 8.

Now all ingredients for a nondeterministic algorithm to compute feasible jobs from a collection of original jobs in a map are provided: The rules of the transformation units *split-freight*, *split-route*, *merge-jobs*, and *check* may be applied in any order and as long as one likes. This is exactly what makes up the transformation unit *compute* shown in Figure 9. If we plug it into the unit *schedule*(*JOBS*) given in Figure 4, we obtain the following semantics of that unit.

> A pair (M, M') of maps (with jobs) is in the semantics of *schedule*(*JOBS*) if and only if the underlying map of M' is M and in M', all jobs in *JOBS* are subdivided in feasible jobs.

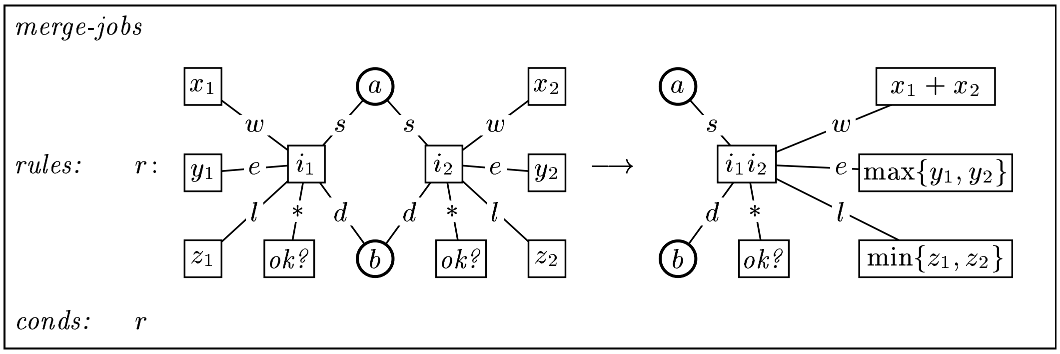


Fig. 8. The transformation unit *merge-jobs*

compute	
uses:	*check*, *split-freight*, *split-route*, *merge-jobs*
conds:	(*split-freight* | *split-route* | *merge-jobs* | *check*)*

Fig. 9. The transformation unit *compute*

5 Extension to the multi-depot problem

The PDP with maximal load and time windows is extended to a multi-depot problem with a limited vehicle fleet by distinguishing some locations as depots, i.e. in such a location there is a number of vehicles available to execute the jobs. Moreover, any tour of a vehicle is required to be a circuit, meaning that the vehicle must start from and return to its home depot. In addition, we assume that every circuit must be covered within a certain time frame *maxt*, e.g. a driver's maximal steering time.

A representation of a vehicle can be added to a map by applying the transformation unit *add-vehicle* shown in Figure 10. It is similar to *insert-job* in that a vehicle with identity v is added whose earliest departure time is *start*, but its final destination is the source location *loc* (the depot), the vehicle does not yet carry any weight as the set of loaded jobs is $\emptyset$, and the latest arrival time automatically computes to $start + maxt$.

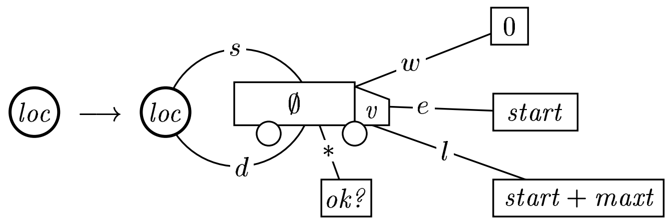
In analogy to the transformation unit *schedule*, *schedule2* in Figure 11 specifies the PDP with depots and limited vehicle fleet: First the available vehicles are added to the map, then the jobs, and after that a solution is computed.

For the computation of a feasible schedule, the transformation unit *compute2* shown in Figure 12 reuses the units *split-freight* and *split-route* introduced in Section 4 to break the original jobs down into feasible subjobs.

An intermediary stop is added to the tour of a vehicle by a transformation unit *add-stop* that has a rule analogous to the one from *split-route*, and a second rule differing from the first only in that the source and destination of the vehicle are the same location. This second rule is needed for the technical reason that

add-vehicle(v,*loc*,*start*)

rules: r:



conds: r

Fig. 10. The transformation unit *add-vehicle*

*schedule*2(*VEHICLES*,*JOBS*)

initial: map

uses: *add-vehicle*, *insert-job*, *compute2*

conds: for each $v \in$ *VEHICLES*: *add-vehicle*($v, loc_v, start_v$);
for each $id \in$ *JOBS*: *insert-job*($i, wt_i, src_i, dst_i, edt_i, lat_i$);
compute2

terminal: no *ok?*

Fig. 11. The transformation unit *schedule2*

our graph transformation approach uses injective matching (cf. [HPM01]). Using *add-stop* means applying one of the rules once.

Replacing the merging of two jobs, a vehicle traveling from a to b loads a job with the same source and destination as shown in Figure 13: The job is added to the set of previously loaded jobs, and the total load, earliest departure and latest arrival time of the vehicle have to be recomputed.

compute2

uses: *split-freight*, *split-route*, *add-stop*, *load-job*, *check-vehicle*

conds: (*split-freight* | *split-route* | *add-stop* | *load-job* | *check-vehicle*)*

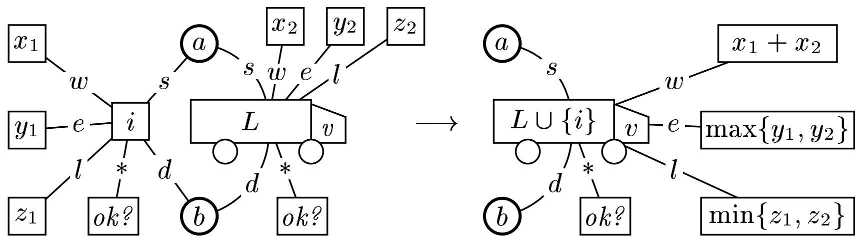
Fig. 12. The transformation unit *compute2*

Finally, instead of the jobs the loaded vehicles are checked for feasibility. This is done by a transformation unit *check-vehicle* analogous to *check*.

The interested reader may find a solution to the running example with a fleet of three vehicles U, V, W, with respective home depots A, C, C, start times 8, 6, and 10, and given *maxt* and *maxwt*.

load-job

rules: r:



conds: r

Fig. 13. The transformation unit *load-job*

6 Conclusion

In this paper we have presented a case study which models the Pickup-and-Delivery Problem with graph transformation units. The problem domain consisting of a map and a set of jobs with certain constraints has been represented as a graph, and the fundamental operations of the algorithms such as the splitting and merging of jobs have been represented as graph transformation rules. The application of the rules is equivalent to the execution of the corresponding operations. In this way we have illustrated that graph transformation provides a means to visualize the PDP. We believe that in general such visualizations may help to better understand the problem and the functionality of the algorithms.

The algorithm modeled in the case study is highly nondeterministic because the graph transformation rules can be applied in almost any order and acceptable solutions are separated from unacceptable ones via graph class expressions. In this sense the presented transformation units illustrate the basic operations of the problem solutions, but more sophisticated optimization strategies should be incorporated in the future. Nevertheless, as mentioned before, the results may serve as input for the *k-opt* algorithm (cf. [LK73]) or as first generation for a *genetic algorithm* as described in [Mic96]. Moreover, an interesting research topic is to investigate the suitability of graph transformation for further or more specific logistic problems. Our case study can be extended in mainly three ways.

1. To make the scenario more realistic, the geographical information can be enhanced e.g. by one-way streets, tunnels, predefined consolidation centers, or stocks.
2. Concerning the transportation we can distinguish different kinds of freight, like containers, hazardous goods, liquids, etc., demanding for different additional constraints. Loading and unloading times can be embodied in the computation. An algorithm might also deal with priorities among the customers or destinations.
3. The given transportation modes are of interest. We can consider homogenous fleets consisting of one type of vehicle, or inhomogeneous fleets. Capacities, i.e. weight or volume, may be restricted. A difficult problem is to schedule multi-modal transports, e.g. transports partly by train and by truck.

All these extensions can be implemented by modifying the underlying class of graphs and the transformation units in a straightforward way. We expect that similar logistic problems are equally amenable to specification by structured graph transformation. Therefore our approach may yield a general methodology for the modelization of logistic problems, which in turn may serve as a basis to develop a visual language for logistic problems. Another interesting point is the use of agent systems, which gives us the possibility to describe the algorithms in a concurrent way. Currently, we are working out a case study which models a graph transformation-based and concurrent solution of the PDP.

References

[CEH+97] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg [Roz97].

[DKKK00] Frank Drewes, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. Graph transformation modules and their composition. In M. Nagl, A. Schürr, and M. Münch, editors, *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, volume 1779 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2000.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* World Scientific, Singapore, 1999.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution.* World Scientific, Singapore, 1999.

[Eul36] L. Euler. Solutio problematis ad geometriam situs pertinensis. *Comment. Academiae Sci.I.Petropolitanae*, 8:128–140, 1736. Opera omnia Ser.I, Vol.7 (1766), 1–10. German translation in: Speiser, Klassische Stücke der Mathematik. Zürich 1927, S.127–138.

[HEET99] Reiko Heckel, Gregor Engels, Hartmut Ehrig, and Gabriele Taentzer. Classification and comparison of module concepts for graph transformation systems. In Ehrig et al. [EEKR99], pages 639–689.

[HPM01] Annegret Habel, Detlef Plump, and Jürgen Müller. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11:637–688, 2001.

[KK99] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[Kus00] Sabine Kuske. *Transformation Units—A structuring Principle for Graph Transformation Systems.* PhD thesis, University of Bremen, 2000.

[Kus01] Sabine Kuske. Parameterized transformation units. In *Proc. GETGRATS Closing Workshop*, volume 51 of *Electronic Notes in Theoretical Computer Science*, 2001. To appear.

[LK73] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21(9):498–516, 1973.

[Mic96] Zbigniew Michalewicz. *Genetic Algorithms + Datastructures = Evolution Programs.* Springer, 3rd edition, 1996.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific, Singapore, 1997.

[SS95] M.W.P. Savelsbergh and M. Sol. The general pickup and delivery problem. *Transportation Science*, 29(1):17–29, 1995.

[Wil70] N.H.M. Wilson. *Dynamic Routing: A Study of Assignment Algorithms.* PhD thesis, Massachusetts Institute of Technology, Department of Civil Engineering, 1970.

Towards Graph Transformation with Time

Szilvia Gyapay[1] and Reiko Heckel[2]

[1] Dept. of Measurement and Information Systems, Budapest University of Technology and Economics
H-1521 Budapest, Hungary
gyapay@mit.bme.hu
[2] Dept. of Math. and Comp. Science, University of Paderborn
D-33095 Paderborn, Germany
reiko@upb.de

Abstract. Following TER nets, an approach to the modelling of time in high-level Petri nets, we propose a model of time within (attributed) graph transformation systems where time stamps are represented as distinguished node attributes. Corresponding axioms for the time model in TER nets are generalised to graph transformation systems and semantic variations are discussed.
The resulting notions of *typed graph transformation with time* specialise the algebraic double-pushout (DPO) approach to typed graph transformation. In particular, the concurrency theory of the DPO approach can be used in the transfer of the basic theory of TER nets.

1 Introduction

Recently, a number of authors have advocated the use of graph transformation as a semantic framework for visual modelling techniques both in computer science and engineering (see, e.g., the contributions in [CH00,BPT01]). In many such techniques, the modelling of time plays a relevant role. In particular, techniques for embedded and safety critical systems make heavy use of concepts like timeouts, timing constraints, delays, etc., and correctness with respect to these issues is critical to the successful operation of these systems. At the same time, those are exactly the systems where, due to the high penalty of failures, formally based modelling and verification techniques are most successful. Therefore, neglecting the time aspect in the semantics of visual modelling techniques, we disregard one of the crucial aspects of modelling.

So far, the theory of graph transformation provides no support for the modelling of time in a way which would allow for *quantified* statements like "this action takes 200ms of time" or "this message will only be accepted within the next three seconds", etc. However, from a more abstract, *qualitative* point of view we can speak of temporal and causal ordering of actions thus abstracting from actual clock and timeout values. Particularly relevant in this context is the theory of concurrency of graph transformation, see [Kre77,CMR96,Bal00] or [BCE$^+$99] for a recent survey.

It is the objective of this paper to propose a quantitative model of time within graph transformation which builds on this more abstract qualitative model. Therefore, we will not add time concepts on top of an existing graph transformation approach, but we show how, in particular, typed graph transformation systems in the double-pushout (DPO) approach [CMR96] can be extended *from within* with a notion of time. This allows both the straightforward transfer of theoretical results and the reuse of existing tools.

The following section outlines our approach of the problem.

2 From Nets to Graph Transformation, with Time

When trying to incorporate time concepts into graph transformation, it is inspiring to study the representation of time in Petri nets. Nets are formally and conceptually close to graph transformation systems which allows for the transfer of concepts and solutions. This has already happened for relevant parts of the concurrency theory of nets which, as mentioned above, provides a qualitative model of time based on the causal ordering of actions.

In particular, we will follow the approach of time ER nets [GMMP91]. These are simple high-level nets which introduce time as a distinguished data type. Then, time values can be associated with individual tokens, read and manipulated like other token attributes when firing transitions. In order to ensure meaningful behaviour (like preventing time from going backwards) constraints are imposed which can be checked for a given net. The advantage of this approach with respect to our aims is the fact that time is modelled within the formalism rather than adding it on top as a new formal concept.

Based on the correspondence of Petri nets and (typed) graph transformation, which regards Petri nets as rewriting systems on multi-sets of vertices [CM95], we can derive a model of time within typed graph transformation systems with attributes. The correspondence is visualised in Table 1. Besides (low-level) place-transition nets and typed graph transformation systems, it relates (high-level) environment-relationship nets to typed graph transformation with attributes. This relationship, which has first been observed in the case of algebraic high-level nets [EPR94] and attributed graph transformation [LKW93] in [Rib96], shall enable us to transfer the modelling of time in time ER nets to typed graph transformation with attributes.

Table 1. Corresponding Petri net and graph transformation variants

	Petri nets	graph transformation systems
low-level	PT nets	typed graph transformation (TGT)
high-level	ER nets	typed graph transformation with attributes (TGTA)
with time	TER nets	typed graph transformation with time (TGTT)

Next, we review time environment-relationship (TER) nets [GMMP91] in order to prepare for the transfer to typed graph transformation systems in Section 4.

3 Modelling Time in Petri Nets

There are many proposals for adding time to Petri nets. In this paper we concentrate on one of them, time ER nets [GMMP91], which is chosen for its general approach of considering time as a token attribute with particular behaviour, rather than as an entirely new concept. As a consequence, time ER nets are a special case of ER nets.

3.1 ER nets

ER (environment-relationship) nets are high-level Petri nets (with the usual net topology) where tokens are environments, i.e., partial functions $e : ID \to V$ associating attribute values from a given set V to attribute identifiers from a given set ID. A marking m is a multi-set of environments (tokens).

To each transition t of the net with pre-domain $p_1 \ldots p_n$ and post-domain $p'_1 \ldots p'_m$, an action $\alpha(t) \in Env^n \times Env^m$ is associated. The projection of $\alpha(t)$ to the pre-domain represents the firing condition, i.e., a predicate on the tokens in the given marking which controls the enabledness of the transition. If the transition is enabled, i.e., in the given marking m there exist tokens satisfying the predicate, the action relation determines possible successor markings.

Formally, a transition t is enabled in a marking m if there exists a tuple $\langle pre, post\rangle \in \alpha(t)$ such that $pre \leq m$ (in the sense of multiset inclusion). Fixing this tuple, the successor marking m' is computed, as usual, by $m' = (m - pre) + post$, and this firing step is denoted by $m[t(pre, post)\rangle m'$. A firing sequence of $s = m_0[t_1(pre_1, post_1)\rangle \ldots [t_{k-1}(pre_{k-1}, post_{k-1})\rangle m_k$ is just a sequence of firing steps adjacent to each other.

3.2 Time ER nets

Time is integrated into ER nets by means of a special attribute, called *chronos*, representing the time of creation of the token as a time stamp. Constraints on the time stamps of both given tokens and tokens that are produced can be specified by the action relation associated to transitions. To provide a meaningful model of time, action relations have to satisfy the following axioms with respect to *chronos* values [GMMP91].

Axiom 1: Local monotonicity For any firing, the time stamps of tokens produced by the firing can not be smaller than time stamps of tokens removed by the firing.

Axiom 2: Uniform time stamps For any firing $m[t(pre, post)\rangle m'$ all time stamps of tokens in *post* have the same value, called the *time of the firing*.

Axiom 3: Firing sequence monotonicity For any firing sequence s, firing times should be monotonically nondecreasing with respect to their occurrence in s.

The first two axioms can be checked locally based on the action relationships of transitions. For the third axiom, it is shown in [GMMP91] that every sequence s where all steps satisfy Axioms 1 and 2 is *permutation equivalent* to a sequence s' where also Axiom 3 is valid. Here, permutation equivalence is the equivalence on firing sequences induced by swapping independent steps. Thus, any firing sequence can be viewed as denoting a representative, which satisfies Axiom 3.

It shall be observed that TER nets are a proper subset of ER nets, i.e., the formalism is not extended but specialised. Next, we use the correspondence between graph transformation and Petri nets to transfer this approach of adding time to typed graph transformation systems.

4 Modelling Time in Graph Transformation Systems

Typed graph transformation systems provide a rich theory of concurrency generalising that of Petri nets [BCE+99]. In order to represent time as an attribute value, a notion of typed graph transformation with attributes is required. We propose an integration of the two concepts (types and attributes) which presents attribute values as vertices and attributes as edges, thus formalising typed graph transformation with attributes as a special case of typed graph transformation.

Next, we give a light-weight (set-theoretic) presentation of the categorical DPO approach [EPS73] to the transformation of typed graphs [CMR96].

4.1 Typed graph transformation

In typed graph transformation, graphs occur at two levels: the type level and the instance level [CMR96]. A fixed *type graph* TG (which may be thought of as an abstract representation of a class diagram) determines a set of *instance graphs* $\langle G, g : G \to TG \rangle$ which are equipped with a structure-preserving mapping g to the type graph (formally expressed as a *graph homomorphism*).

A *graph transformation rule* $p : L \to R$ consists of a pair of TG-typed instance graphs L, R such that the union $L \cup R$ is defined. (This means that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.) The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

A *graph transformation* from a pre-state G to a post-state H, denoted by $G \stackrel{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and

– $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

A transformation sequence $G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n$ is a sequences of consecutive transformation steps.

On transformation sequences, a notion of equivalence is defined which generalises the permutation equivalence on firing sequences: two sequences are equivalent if they can be obtained from each other by repeatedly swapping independent transformation steps. This equivalence has been formalised by the notion of *shift-equivalence* [Kre77] which is based on the following notion of independence of graph transformations. Two transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ are *independent* if the occurrences $o_1(R_1)$ of the right-hand side of p_1 and $o_2(L_2)$ of the left-hand side of p_2 do only overlap in objects that are preserved by both steps, formally $o_1(R_1) \cap o_2(L_2) \subseteq o_1(L_1 \cap R_1) \cap o_2(L_2 \cap R_2)$. This is more sophisticated than the notion of independent firings of transitions which are required to use entirely disjoint resources.

4.2 Typed graph transformation with attributes

Assuming a set of data type symbols S, a *type graph with attribute declarations* (based on S) is a graph TG whose set of vertices TG_V contains S. Therefore, data type symbols are vertex types so that edges, representing attribute declarations, may be drawn towards them from ordinary vertices. This is compatible with notions of attributed graphs, like [LKW93], where attribute carriers are used to relate graph elements and data values. Notice, however, that we limit ourselves to attributed vertices, and that we do not extend but refine the notion of graph.

Given a data domain D_s for every data type symbol s, an *instance graph with attributes* over the type graph TG is an instance graph $\langle G, g : G \to TG \rangle$ over TG (in the above sense) such that $g^{-1}(s) = D_s \subseteq G_V$. Therefore, all vertices $x \in G_V$ with $g(x) \in S$ represent attribute values which may or may not be referenced by an edge from another vertex. As a consequence, instance graphs will be usually infinite. E.g., if the data type $\mathbb{N}$ of natural numbers is present, each $n \in \mathbb{N}$ will be a separate vertex.

Morphisms between instance graphs with attributes $\langle G, g : G \to TG \rangle$ and $\langle H, h : H \to TG \rangle$ are typed graph morphisms $f : G \to H$, i.e., graph morphisms compatible with the typing ($h \circ f = g$) and preserving the data domains; formally $f|_S = id_{G|_S}$, if we denote by $f|_S : G|_S \to H|_S$ the restriction of f_V to vertices $x \in V$ of type $g(x) \in S$.

From this point on, all other notions, like rule, occurrence, transformation, transformation sequence, etc. are defined as in the previous subsection. Also, relevant results like the Local Church-Rosser Theorem, the Parallelism theorem, and the corresponding equivalence on transformation sequences based on shifting or swapping of independent transformations are easily transferred.

It is worth noticing that, in contrast with ER nets, attributes in our model are typed, that is, different types of nodes may have different selections of attributes.

However, like in ER nets, our data types have no syntax: We only consider sets of values without explicit algebraic structure given by operations. As a consequence, we do not explicitly represent variables within rules and variable assignments as part of occurrences: A rule containing variables for attribute calculation and constraints is considered as a syntactic representation of the (possibly infinite) set of its instances where the variables and expressions are replaced by concrete values.

4.3 Typed graph transformation with time

To incorporate time into typed graph transformation with attributes, we follow the approach of TER nets as discussed in Section 3. Therefore, a time data type is required as domain for time-valued attributes.

A *time data type* $T = \langle D_{time}, +, 0, \geq \rangle$ is a structure where $\geq$ is a partial order with 0 as its least element. Moreover, $\langle +, 0 \rangle$ form a monoid (that is, $+$ is associative with neutral element 0) and $+$ is monotone wrt. $\geq$. Obvious examples include natural or real numbers with the usual interpretation of the operations, but not dates in the YY:MM:DD format (e.g., due to the Y2K problem).

A *type graph with time* TG is a type graph with attribute declarations based on a set of data type symbols S that contains a special symbol $time$. An instance graph with time over TG for a given time data type $T = \langle D_{time}, +, 0, \geq \rangle$ is an instance graph $\langle G, g : G \to TG \rangle$ over TG such that the data type sort $time$ is interpreted by D_{time}, that is, $D_{time} = \{x \in G_V | g(x) = time\}$. Graph morphisms are defined as before.

The definition of graph transformation rules with time has to take into account the particular properties of time as expressed, for example, by the axioms in Section 3. Due to the more general nature of typed graph transformation in comparison with ER nets, there exist some degrees of freedom in the transfer of these axioms.

First, ER nets are untyped (that is, all tokens have (potentially) the same attributes) while in typed graph transformation we can declare dedicated attributes for every vertex type. Therefore, we do not have to assume a uniform time attribute *chronos*, but could leave the choice of time attributes to the designer. In particular, this would allow a vertex type not to have time stamps at all, or to have several different time stamps, e.g., one for its creation and one for the last update. Here, we consider time as a distinguished semantic concept which should not be confused with time-valued data. Therefore, we decide for the uniform treatment of time stamps using *chronos*. (This does not forbid us to model additional time-valued data by ordinary attributes.)

The second degree of freedom comes from the (well-known) fact that graph transformations generalise Petri nets by allowing contextual rewriting: All tokens in the post-domain of a transition are newly created while in the right-hand side of a graph transformation rule there may be vertices that are preserved. This allows different generalisations of Axiom 2: The extremes are (1) to require uniform time values only for new vertices or vertices whose *chronos* values are updated, or (2) to assign new uniform chronos values to all vertices in the right-hand side. For the moment, we decide for the second, more conservative variant.

Therefore, we define a typed graph transformation rule with time as a graph transformation rule with attributes such that

Condition 1. Local monotonicity: for all vertices $x \in L$ and $y \in R$, the time stamp of x is smaller or equal to the time stamp of y, and

Condition 2. Uniform time stamps: for all vertices $x, y \in R$ the time stamp of x equals the time stamp of y.

These conditions ensure a behaviour of time which can be described informally as follows. According to condition 1 an operation or transaction specified by a rule cannot take negative time, i.e., it cannot decrease the time stamps of the nodes it is applied to. Condition 2 states an assumption about atomicity of rule application, that is, all effects specified in the right-hand side are observed at the same time, called the firing time of the rule.

In this case we can show, in analogy with TER nets, that for each transformation sequence s using only rules that satisfy the above two conditions, there exists an equivalent sequence s' such that s' is time-ordered, that is, time stamps are monotonically non-decreasing as the sequence advances. This is no longer true if we use the more liberal interpretation of Axiom 2.

Condition 2'. Uniform time stamps for new or updated objects: for all vertices $x, y \in R$ who are newly created or whose time stamps are updated, the time stamp of x equals the time stamp of y.

In this case, it is no longer true that for all transformation sequences using rules that satisfy Conditions 1 and 2', there exists an equivalent time-ordered sequence. This is shown by the following counterexample.

Example. Figure 1 shows a type graph TG and a (generic) instance graph IG of TG, respectively. The type graph defines three vertex types: $T1, T2$ and $T3$. For $T2$ and $T3$ *chronos* attributes are declared, while $T1$ has no attribute. (Our example does not need edges.)

The instance graph contains nodes $A : T1$ (i.e., a $T1$-typed node named A), $B : T2$, and $C : T3$, where C has the chronos value $c2$ and the chronos value of B is $c2 + 3$. Two rules, p_1 and p_2, are defined in Figure 2. By applying p_1, nodes $a : T1$ and $b : T2$ are matched and the chronos value of b is increased by 4 time units.

Rule p_2 requires nodes $a : T1$ and $b : T3$. The former is deleted and the time of the latter is increased by 2 units. (Note, that the use of similar names does not imply any connection between the elements of different rules.)

An application of these rules to the instance graph in Figure 1 is shown in Figure 3. Both p_1 and p_2 are applicable to the graph. Applying first p_1 and then p_2 leads to the graph in the lower right. In this sequence, first the chronos value of B is increased and then A is deleted and the chronos value of C is increased. The occurrences and the firing times of the steps are denoted next to the arrows.

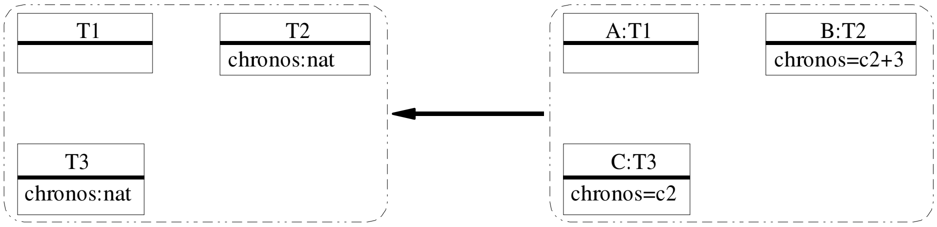


Fig. 1. Type graph (left) and instance graph (right)

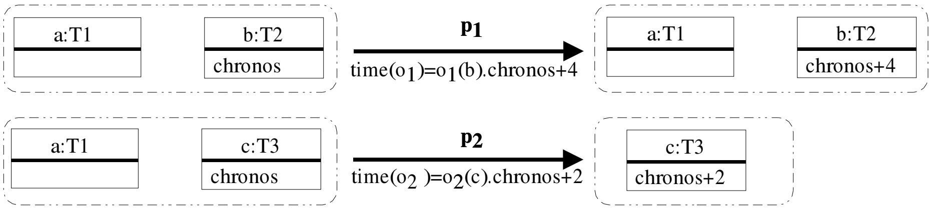


Fig. 2. Rules p_1 and p_2

At this point, two observations are crucial. First, the two steps are not independent, that is there exists no equivalent sequence where p_1 and p_2 are applied in the reverse order. This is because $A \in (o_1(L_1 \cap R_1)) \cap o_2(L_2 \setminus R_2))$, i.e., A is deleted by p_2 but required by p_1. Second, the sequence $IG \stackrel{p_1(o_1)}{\Longrightarrow} IG_1 \stackrel{p_2(o_2)}{\Longrightarrow} IG_2$ is not time-ordered because the firing time of the latter is smaller than the firing time of the first.

Note that, if A would have a chronos attribute *and* all chronos values would be updated uniformly as required by Condition 2, A should get $time(o_1)$ thus disabling its deletion at a lower time.

Conceptually, Axiom 3 means to assume (or be able to establish) global time, which is not always realistic, in particular when considering asynchronous (e.g., wide-area) networks. Therefore, also the more liberal interpretation may be justified even if it has weaker semantic properties. In particular, notice that it allows for a higher degree of concurrency because, due to the update of *chronos* values on all nodes in the right-hand side, in the conservative variant independence of transformations is reduced to disjoint matches.

5 Conclusion

We have transferred the model of time within ER nets, a kind of high-level Petri nets, to graph transformation systems. The resulting notion is a special case of typed graph transformation, where certain vertices are interpreted as time values

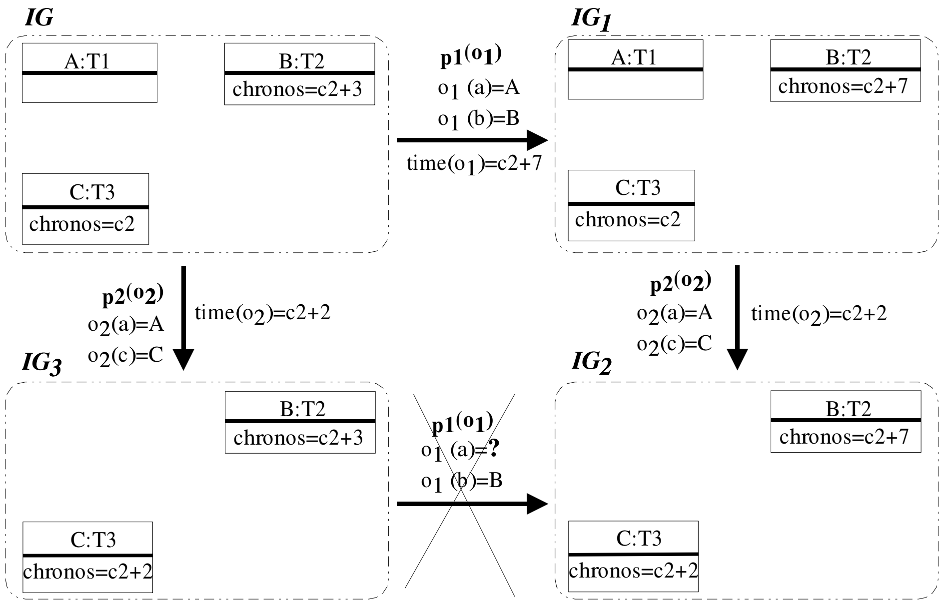


Fig. 3. Rule sequence

and edges towards these vertices are time-valued attributes. We have discussed some choices and their semantic consequences.

It requires a deeper analysis of potential applications, in particular, the use of time in diagrammatic techniques like statecharts or sequence diagrams and their existing formalisations within graph transformation [EHHS00,HHS01,Kus01] to understand if these choices are the right ones.

Acknowledgement. We wish to thank Luciano Baresi for his introduction to Petri nets with time, and to Daniel Varro for pointing out semantic alternatives.

References

[Bal00] P. Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 2000.

[BCE+99] P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Concurrent semantics of algebraic graph transformation. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*, pages 107 – 188. World Scientific, 1999.

[BPT01] L. Baresi, M. Pezzé, and G. Taentzer, editors. *Proc. ICALP 2001 Workshop on Graph Transformation and Visual Modeling Techniques, Heraklion, Greece*, Electronic Notes in TCS. Elsevier Science, July 2001.

[CH00] A. Corradini and R. Heckel, editors. *Proc. ICALP 2000 Workshop on Graph Transformation and Visual Modeling Techniques*, Geneva, Switzerland, July 2000. Carleton Scientific. http://www.di.unipi.it/GT-VMT/.

[CM95] A. Corradini and U. Montanari. Specification of Concurrent Systems: from Petri Nets to Graph Grammars. In *Quality of Communication-Based Systems*, pages 35–52. Kluwer Academic Publishers, 1995.

[CMR96] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.

[EHHS00] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000, York, UK*, volume 1939 of *LNCS*, pages 323–337. Springer-Verlag, 2000.

[EPR94] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic high-level nets: Petri nets revisited. In *Recent Trends in Data Type Specification*, pages 188–206, Caldes de Malavella, Spain, 1994. Springer Verlag. Lecture Notes in Computer Science 785.

[EPS73] H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[GMMP91] C. Ghezzi, D. Mandrioli, S. Morasca, and Pezzè. A unified high-level petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, 1991.

[HHS01] J.H. Hausmann, R. Heckel, and S. Sauer. Towards dynamic meta modeling of UML extensions: An extensible semantics for UML sequence diagrams. In M. Minas and A. Schürr, editors, *Symposium on Visual Languages and Formal Methods, IEEE Symposia on Human Computer Interaction (HCI 2001), Stresa, Italy*, Los Alamitos, CA, September 2001. IEEE Computer Society Press.

[Kre77] H.-J. Kreowski. *Manipulation von Graphmanipulationen.* PhD thesis, Technical University of Berlin, Dep. of Comp. Sci., 1977.

[Kus01] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001, Toronto, Kanada*, volume 2185 of *LNCS*. Springer-Verlag, 2001.

[LKW93] 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.

[Rib96] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, TU Berlin, 1996.

Modeling Hyperweb Dynamics through Hierarchical Graph Transformation

Giorgio Busatto

Department of Computer Science,
Carl v. Ossietzky University,
Oldenburg, Germany
Giorgio.Busatto@Informatik.Uni-Oldenburg.DE

1 Introduction

Hypermedia systems are very popular in information technology (consider e.g. the World-Wide Web) since they provide an intuitive paradigm for organizing and accessing large collections of multimedia documents (*hyperwebs*).

With the increasing complexity of hypermedia applications, it is more and more important to have specification techniques that are on a high level of abstraction and formal enough to prove properties of the described applications. Existing models (e.g. [HS94], [SRB96], [TEP95], [Mau96]) concentrate on static structural aspects offering high-level primitives like indices, hierarchical navigation spaces, and so on. Even though some approaches address dynamic aspects of hypermedia (e.g. synchronization between different documents, see [SDW95]), conceptual models of hyperweb transformation (*hyperweb dynamics*) are still missing, forcing developers to reason about these aspects at a low implementation level (see [BtH00]).

Looking for a conceptual model of hyperweb transformation, we consider graphs and (rule-based) graph transformation. First, hyperwebs can be easily interpreted as graphs: they are networks of documents and links. Second, graph transformation (see e.g. [Roz97], [EEKR99], [EKMR99]) provides an intuitive, yet formal, way of specifying operations on graphs. Third, there exist several approaches that use *hierarchical graphs*, with support for grouping and encapsulation of nodes and edges (see e.g. [Pra79], [ES95], [BEMW00], [EH00], [DHP02], [BH01]), thus addressing the need for structuring in hypermedia. These factors make hierarchical graph transformation very appealing for our purposes.

In order to evaluate the possible impact of hierarchical graph transformation on hyperweb dynamics, we apply these ideas to a concrete hypermedia system, namely HyperWave (see [Mau96]). It turns out that hierarchical graph transformation provides an intuitive and concise way of specifying (complex) hyperweb transformations, and that it is on a comparable level of abstraction as HyperWave's hypermedia model. Furthermore, our approach allows to check statically whether the defined hyperweb manipulations preserve consistency, thus helping to write correct applications and reducing the amount of consistency checks that must be performed at runtime.

This contribution has the following structure. In Section 2, we provide a summary of the HyperWave data model and introduce a running example. In Section 3, we recall the notion of a hierarchical graph and use it to model hyperwebs. In Section 4, we introduce hierarchical graph transformation and use it to specify hyperweb manipulations. In Section 5, we provide a summary and some concluding remarks.

2 Hypermedia and HyperWave

In the HyperWave$^{\mathrm{tm}}$ system a hypermedia *repository*—i.e. a hyperweb—provides *documents* with *referential links* between them. A link starts from an *anchor* attached to a document, and ends at a destination anchor, document, or *collection*. Collections are organizational structures that contain documents and (possibly) other collections[1]. Documents and collections can be shared by other collections, but the collection hierarchy must be cycle free (dag). All documents and collections (except the *root collection*) must be contained in at least one collection (*orphans* are forbidden). Empty repositories are forbidden.

Consider the following example scenario. A company stores its employees' documents in a repository. Each user has a home collection, with work-related documents, and some personal documents. A project manager has a "projects" collection, a "phone" document, a personal collection, and an "archive" collection about completed projects. Figure 1 shows the contents of a project manager's collection. Collections are drawn as rectangles with tabs, documents as rectangles, anchors as small black boxes. Arrows between collections indicate their nesting, arrows from collections to documents indicate that a collection contains that document, anchors are attached to pages using undirected lines, and hyperlinks are drawn as arrows originating from anchors.

The company has several classes of users, with corresponding home collection structures. An administrator's collection (see Figure 3) contains a "staff" and a "financial report" document, and no "projects" collection in the top level of its hierarchy. If, say, a manager becomes an administrator, his or her information structure is reorganized automatically by an application. In this example, the new documents "staff" and "financial report" are added, and the projects are moved to the "archive" collection.

In HyperWave such transformations are programmed as low-level database transactions (see [Mau96, Appendix F]). In Section 4, we show how to specify these manipulations through hierarchical graph transformation.

3 Hyperwebs and Hierarchical Graphs

In this section, we model hyperwebs in HyperWave using the hierarchical graph approach described in [BH01] and [BKK01]. In this approach, a hierarchical graph H consists of the following components:

[1] Besides collections, HyperWave supports other organizational structures, which are not relevant here.

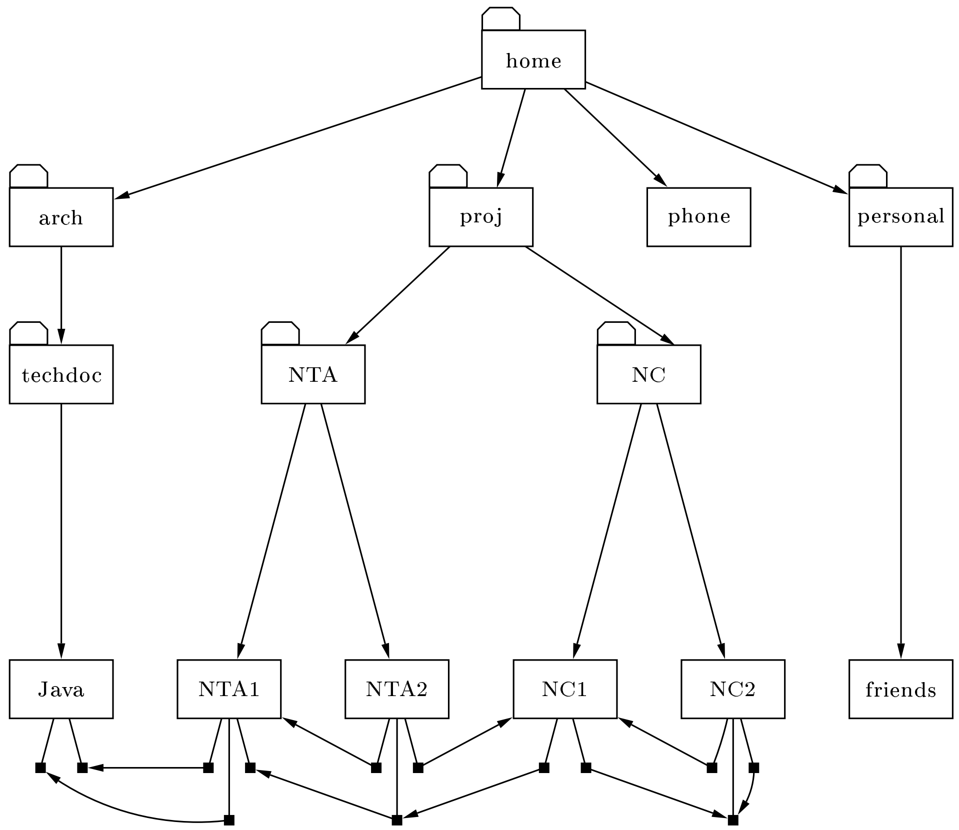


Fig. 1. A project manager's home collection.

- The *underlying graph* U, containing the *nodes* and *edges* of H.
- The *hierarchy graph* P, a rooted directed acyclic graph (rooted dag) that describes the hierarchical structure as a collection of *graph packages* (its nodes) and a parent/child relation (its edges).
- The *coupling graph* C, a bipartite graph connecting nodes of U with nodes of P. Edges from N_U to N_P relate nodes to the package they belong to. Edges from N_P to N_U relate packages to nodes they are *anchored* to.

Anchoring is used when a modeled entity plays both the role of a node and of a grouping component (package). For example, in HyperWave a collection can be thought of both as a node (navigation edges can point to it) and as a package (it groups documents and other collections). Notice that anchoring of packages should not be confused with the use of anchors for hyperlinking within hypermedia.

Hierarchical graphs

A set $\mathcal{G}$ defines a set of graphs if every $G \in \mathcal{G}$ has a *skeleton* $S_G = (N_G, E_G, I_G)$, where N_G and E_G are finite sets of *nodes* and *edges*, and $I_G \subseteq E_G \times N_G$ is an *incidence relation*. Having a skeleton is the minimal requirement for an entity to be considered a graph, and it serves as an interface to the hierarchical structure added to it.

A directed *graph* G consists of disjoint finite sets N_G of *nodes* and E_G of *edges*, with each edge attached to exactly one *source* and one *target* node, and each node (edge) labelled over a given set Σ (resp. Δ) of labels. Every directed graph G provides a skeleton. A *dag* is a cycle-free directed graph. A directed graph is *rooted* if it has a distinguished *root* $\rho \in N_G$ so that there exists a path from ρ to n in G, for all $n \in N_G$. (Rooted dags are used for package hierarchies.)

A bipartite graph C over (M, N)—i.e. a directed graph where all edges have one end in M and the other one in N—is a *coupling graph* if it induces an *association relation* $\hookleftarrow_C \subseteq N \times M$ that assigns every node of M to at least one node of N. A coupling graph C is *tight* if it also induces a *correspondence relation* $\approx_C \subseteq M \times N$ that anchors every node of N at a unique node of M. (Coupling graphs are used for connecting package graphs and underlying graphs.)

A *generic hierarchical graph* is a triple $H = (U, P, C)$, with an *underlying graph* U (of any kind, provided it has a skeleton), a rooted, directed acyclic *package graph* P, and a bipartite *coupling graph* C over nodes (N_U, N_P). If C is tight, H is called *tightly coupled*, and *loosely coupled* otherwise.

Modeling hyperwebs

A repository R with root collection ρ is modeled by a hierarchical graph $\mathbf{HG}(R) = (U, P, C)$, where U represents the navigation structure and P the organization structure. Let $\mathbf{coll}(R)$ be the set of collections of R, $\mathbf{doc}(R)$ the set of documents, and $\mathbf{anc}(R)$ the set of anchors. For each $c \in \mathbf{coll}(R) \setminus \{\rho\}$, we let n_c be a node. We also associate a package p_c to every collection c.

The set of nodes N_U is $\{n_c \mid c \in \mathbf{coll}(R) \setminus \{\rho\}\} \cup \mathbf{doc}(R) \cup \mathbf{anc}(R)$, where the three types of nodes are marked with three disjoint label sets $COLL$, DOC, and $ANC = \{anc\}$, respectively. Anchor nodes are attached to document nodes by directed edges. Links are modeled by edges from source anchors to destination anchors, documents, or collections. The two types of edges are marked with labels $\{\sigma, \lambda\}$, respectively.

The set of packages N_P is $\{p_c \mid c \in \mathbf{coll}(R)\}$ and, for each $c, c' \in \mathbf{coll}(R)$, if c is contained in c' we let p_c be a child of $p_{c'}$ in P (edge from c' to c). The package p_ρ is the root of P. In order to have a uniform graph model for U, P, and C, nodes and edges of P carry a trivial $\perp$ label.

The coupling graph C relates N_U with N_P. Every package p_c ($c \neq \rho$) is anchored to the node n_c (edge from p_c to n_c in C). Every document node $d \in \mathbf{doc}(R)$ and all its anchors $a_1, \ldots, a_k \in \mathbf{anc}(R)$ are associated to all packages p_c (edges to p_c in C), such that d is contained by c in R. We let $N_C := N_U \cup N_P$ and use the same labels as in U and P. All edges of C are labeled with $\perp$.

The condition on hierarchical graphs, that requires that all nodes be at least in one package, ensures that there are no orphans. We depict a hierarchical graph with the same notation as in Figure 1. Labels are written inside nodes, e.g. $COLL = \{\text{home}, \text{arch}, \ldots\}$. Edge labels are omitted, since edges can be distinguished by the notation. A similar notation will be used for transformation rules.

4 Hyperweb Dynamics

In this section, we apply hierarchical graph transformation to the specification of hyperweb manipulations. Again, we follow the approach presented in [BH01] and [BKK01]: hierarchical rules are triples of graph transformation rules that are applied to the components of a hierarchical graph. Using the results proven in [BKK01], we can characterize the rules that manipulate hyperwebs in a consistent way (see below). Such consistency depends on properties of rules that can be checked statically.

Hierarchical graph transformation

The notion of a *graph transformation approach* has been formalized in [AEH$^+$99] in order to specify the common features of as many kinds of graph transformation as possible. Here we are interested in *basic* graph transformation approaches $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow)$ where $\mathcal{G}$ is a class of *graphs*, $\mathcal{R}$ is a class of *rules*, and $\Rightarrow$ is a *rule application operator* that associates a binary relation $\Rightarrow \subseteq \mathcal{G} \times \mathcal{G}$ to every rule $r \in \mathcal{R}$. We omit control conditions and graph class expressions that are used for programming and specification in [AEH$^+$99].

A basic *hierarchical graph transformation approach* $\mathcal{A}_{\mathcal{H}} = (\mathcal{H}, \mathcal{R}_{\mathcal{H}}, \Rightarrow_{\mathcal{H}})$ is constructed by combining an underlying graph transformation approach $\mathcal{A}_u$ over graphs $\mathcal{G}_u$ with two graph transformation approaches $\mathcal{A}_p$ over rooted dags, and $\mathcal{A}_c$ over coupling graphs, respectively, by componentwise composition. If its component approaches have the same application operator, we call $\mathcal{A}_{\mathcal{H}}$ *homogeneous.* The classes of graphs and rules are defined as the Cartesian products of the corresponding component classes, and their semantics is constructed componentwise, too. The application operator is defined as $\Rightarrow_{\mathcal{H}} = \{((U, P, C), (U', P', C')) \in \mathcal{H} \times \mathcal{H} \mid U \Rightarrow_u U', P \Rightarrow_p P', C \Rightarrow_c C'\}$.

We consider a homogeneous hierarchical graph transformation approach using the double pushout approach (DPO, see [CMR$^+$97]). DPO rules are of the form $r = L \overset{i}{\leftarrow} K \overset{j}{\rightarrow} R$ where i and j are graph morphisms mapping an interface graph K to the left-hand side L and right-hand side R of the rule, respectively (i must be injective). A transformation step from a graph G is performed by finding an occurrence of L in G (a morphism $m : L \to G$), building a graph D by removing $m(L)$ from G up to the interface $m(i(K))$, and adding (a copy) of R by identifying all elements of R and D that have a common preimage w.r.t. $m \circ i$ and j, respectively.

In the hierarchical case we will have triples of rules

$$r = (L_u \overset{i_u}{\leftarrow} K_u \overset{j_u}{\rightarrow} R_u, L_p \overset{i_p}{\leftarrow} K_p \overset{j_p}{\rightarrow} R_p, L_c \overset{i_c}{\leftarrow} K_c \overset{j_c}{\rightarrow} R_c)$$

that are applied in parallel to the components of a hierarchical graph. We consider *glued rules*, i.e. rules such that: For $X \in \{L, K, R\}$, $N_{X_c} = N_{X_u} \cup N_{X_p}$ and

$N_{X_u} \cap N_{X_p} = \emptyset$, while the pairs $\langle i_u, i_c \rangle$, $\langle i_p, i_c \rangle$, $\langle j_u, j_c \rangle$, $\langle j_p, j_c \rangle$ agree on common nodes in their domains. For hyperweb transformation we use coordinated hierarchical graph transformation, where glued rules are used and the matching morphisms of the component rules overlap on the nodes of the coupling graph. We indicate a coordinated transformation step with $G \Rrightarrow_r G'$.

Modeling hyperweb dynamics

In Figure 2, we show a rule that specifies the transformation of a manager collection to an administrator collection. The left-hand side L_p contains three packages and two hierarchy edges. The three packages are also in the glued left-hand side L_c, while L_u is empty. The right-hand side R_p contains three packages (shared with R_c) and a different hierarchy structure. R_u contains two nodes (shared with R_c). The interfaces are represented by means of identifiers: the notation x' = 'x in the right-hand side denotes a preserved node (a node that is also in the interface). In 'x: y (resp. x': y), y denotes the label of the matched (resp. inserted) node x. This rule hides the project collection inside the archive collection, and adds the two new documents staff and financial report. By applying this rule to the hyperweb of Figure 1, we obtain the hyperweb depicted in Figure 3.

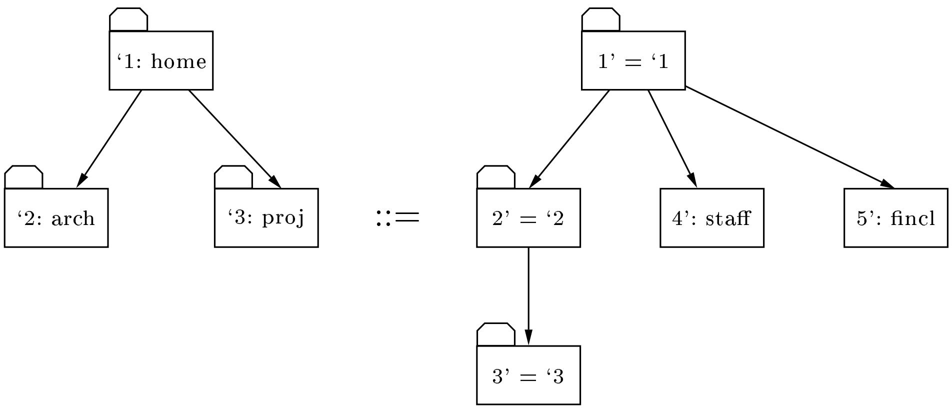


Fig. 2. Reorganizing a collection.

By using the results proven in [BKK01], this approach allows to check *statically* whether the defined transformations preserve consistency inside a repository. In fact

- link consistency is always ensured by DPO rules (no dangling edges are introduced),
- consistent hierarchy transformation (rooted dags are transformed to rooted dags) is ensured by properties of rules that can be checked statically using Proposition 5.20 in [BKK01],

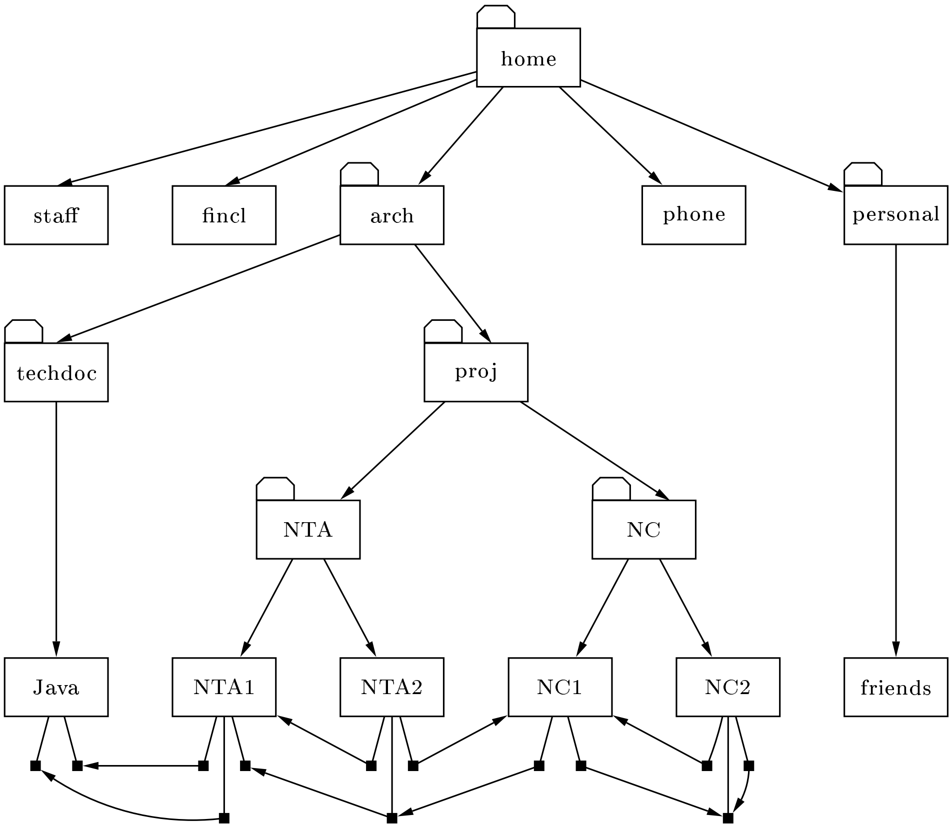


Fig. 3. A administrator's home collection.

– the absence of orphan documents in the transformed graphs is also related to properties of rules that can be checked statically using Proposition 5.12 in [BKK01].

Similar consistency checks are performed by a HyperWave server at *runtime*.

5 Conclusion and Future Work

In this paper, we have applied hierarchical graph transformation to the specification of hyperweb manipulations in the HyperWave system.

Compared to HyperWave, our approach is visual and more concise, and lies at a conceptual level, since it allows to reason in terms of organization and navigation structures, avoiding implementation details like the sequences of commands to be exchanged with a server. Furthermore, by using the double-pushout approach, we can use results about hierarchical graph transformation (see [BKK01]) to check statically whether the defined operations preserve consistency in a hyperweb. Such checks are performed at runtime in HyperWave.

In the HyperView approach (see e.g. [Fau00]), the structure of web pages and web sites is modeled using *clustered graphs*, where a *structure graph* is related to a *base graph* by a *clustering morphism*, thus defining a decoupling approach that has some similarity to ours. HyperView also uses (single-pushout, see [EHK+97]) rules for defining the translation between different views of existing hyperwebs (web sites). Instead, in the present contribution we propose to use rules to modify the content of hyperwebs. A closer comparison of HyperView with our approach can be an interesting topic for future research.

An open issue concerning our approach is related to the efficiency of (hierarchical) graph transformation. First, the graph isomorphism problem—occurring every time a rule is matched w.r.t. a host graph—is NP-complete. There exist approaches (e.g. [BGT91], [MB95], [Rud00]) that try to improve this limit in cases of practical interest. For example, [MB95] introduces an algorithm that solves the problem in polynomial time, provided that the graph to be matched is a-priori known. The impact of these results on our approach is a topic for future research.

Another source of inefficiency is the non-determinism of rule application and the need for backtracking. This situation can be improved by adding control structures to drive rule application, like in PROGRES (see [SWZ99]) and in GRACE (see [Kus99]). *Transformation units* (the concept used in GRACE) are of particular interest because they can be combined with our model of hierarchical graph transformation since both rely on the same notion of a graph transformation approach. The use of transformation units in our approach should therefore be investigated.

Another interesting topic is the use of other approaches than DPO for specifying hyperweb manipulation. For example, imagine that we want to modify a portion of web and update a lot of hyperlinks connecting it to its context in a consistent way. Such situations suggest that a *connecting* approach (e.g. NLC, [ER97]) can be better suited than DPO. Our *generic* hierarchical graph notion allows to change the underlying graph transformation approach smoothly.

Finally, we have not modeled *access permissions* of HyperWave yet, and this suggests that the notion of *encapsulated hierarchical graph* transformation (see [ES95], [BEMW00]) could be investigated. The above considerations are summarized in Table 1.

	HyperWave	*HG transformation*
Abstraction level	Implementation	Conceptual
Style	Textual, verbose	Visual, concise
Consistency	Checked at runtime	Static check possible
Efficiency	Good	To be investigated
Access control	Access permissions	Encapsulated HG

Table 1. Hyperweb dynamics: HyperWave vs hierarchical graph transformation.

References

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, April 1999.

[BEMW00] Giorgio Busatto, Gregor Engels, Katharina Mehner, and Annika Wagner. A framework for adding packages to graph transformation systems. In Ehrig et al. [EEKR00], pages 352–367.

[BGT91] Horst Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammars based on the RETE-matching algorithm. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 174–189, 1991.

[BH01] Giorgio Busatto and Berthold Hoffmann. Comparing notions of hierarchical graph transformation. In Taentzer et al. [TBP01], pages 312–319.

[BKK01] Giorgio Busatto, Hans-Jörg Kreowski, and Sabine Kuske. Abstract hierarchical graph transformation. Technical Report 1, University of Bremen, Bremen, Germany, 2001.

[BtH00] Giorgio Busatto and Pieter Jan 't Hoen. A graph-grammar based approach for the specification of hypermedia application dynamics. In Corradini and Heckel [CH00], pages 403–409.

[CH00] A. Corradini and R. Heckel, editors. *Workshop on Graph Transformation and Visual Modeling, Satellite of ICALP'2000.* Carleton Scientific, 2000.

[CM95] Andrea Corradini and Ugo Montanari, editors. *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1995.

[CMR+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In Rozenberg [Roz97], chapter 3, pages 163–246.

[DHP02] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, to appear, 2002.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation 2 (Specifications and Programming).* World Scientific, Singapore, 1999.

[EEKR00] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764 in *Lecture Notes in Computer Science*. Springer, 2000.

[EH00] G. Engels and R. Heckel. Graph transformation as unifying formal framework for system modeling and model evolution. In Welzl et al. [WMR00], pages 127–150.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In Rozenberg [Roz97], chapter 4, pages 247–312.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation 3 (Concurrency)*. World Scientific, Singapore, 1999.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [Roz97], chapter 1, pages 1–94.

[ES95] Gregor Engels and Andy Schürr. Encapsulated hierarchical graphs, graph types, and meta types. In Corradini and Montanari [CM95].

[Fau00] Lukas C. Faulstich. Using graph transformation techniques for integrating information from the WWW. In Ehrig et al. [EEKR00], pages 382–389.

[HS94] Frank Halasz and Mayer Schwartz. The Dexter hypertext reference model. *Communications of the Association for Computer Machinery*, 37(2):30–39, February 1994.

[Kus99] Sabine Kuske. *Transformation Units – A Structuring Principle for Graph Transformation Systems*. PhD thesis, Fachbereich 3 (Mathematik & Informatik) der Universität Bremen, 1999.

[Mau96] Hermann Maurer. *Hyperwave, The Next Generation Web Solutions*. Addison Wesley Longman, Edinburgh gate, Harlow, Essex, England, 1996.

[MB95] Bruno T. Messmer and H. Bunke. Subgraph isomorphism in polynomial time. Technical Report IAM 95-003, University of Bern, Institute of Computer Science and Applied Mathematics, Bern, Switzerland, 1995.

[Pra79] Terrence W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 389–400, 1979.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation 1 (Foundations)*. World Scientific, Singapore, 1997.

[Rud00] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In Ehrig et al. [EEKR00], pages 238–251.

[SDW95] P. Senac, P. De Saqui-Sannes, and R. Willrich. Hierarchical time stream Petri net: A model for hypermedia systems. *Lecture Notes in Computer Science*, 935:451–470, 1995.

[SRB96] D. Schwabe, G. Rossi, and S. D. J. Barbosa. Systematic hypermedia application design with OOHDM. In *Hypertext '96, Washington, DC, March 16–20, 1996: the Seventh ACM Conference on Hypertext: Proceedings*, pages 116–128, New York, NY, USA, 1996. ACM Press.

[SWZ99] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [EEKR99], chapter 13, pages 487–550.

[TBP01] Gabriele Taentzer, Luciano Baresi, and Mauro Pezzè, editors. *Workshop on Graph Transformation and Visual Modeling, Satellite of ICALP'2001*. Elsevier, 2001.

[TEP95] Tomás Isakowitz, Edward A. Stohr, and P. Balasubramanian. RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, August 1995.

[WMR00] Emo Welzl, Ugo Montanari, and Jose D. P. Rolim, editors. *Automata, languages and programming: 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9–15, 2000: proceedings*, volume 1853 of *Lecture Notes in Computer Science*, New York, NY, USA, 2000. Springer-Verlag Inc.

Transformation Systems for the Integration of Software Specifications *

Martin Große–Rhode, Sebastian John, Gunnar Schröter

Insitut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
email: {mgr,sebc,schroetg}@cs.tu-berlin.de,
URL: http://tfs.cs.tu-berlin.de/~{mgr,sebc,schroetg}

Abstract. In a software system development process a variety of heterogeneous viewpoint models of the system are constructed for the abstract specification of its functionality and behaviour. These have to be integrated in order to achieve a coherent and consistent global system specification. Transformation systems constitute a formal semantic domain where specification languages conforming to different paradigms can be interpreted, which makes possible a formal semantic integration of heterogeneous viewpoint models. Analogous to graph transformation systems states are described by algebraic structures and behaviour by the ordering of state transformation steps, i.e., changes of the algebraic structures. In contrast with graph transformation systems, however, transformation systems are generic w.r.t. the choice of the state representations, i.e., instead of graphs or extensions thereof arbitrary structures and corresponding logics may be used. Moreover, development relations and structuring operations are defined, which altogether yields a powerful and flexible integration framework.

1 Introduction

In the development process of large software systems usually a series of different models of the system is constructed. These range from first sketches of the system and its environment (the domain) through to detailed design models. Thereby different aspects of the system and its components are specified from different points of view. Following the *viewpoint model* of software systems development, as promoted in the Reference Model of Open Distributed Processing [14] for example, different languages should be used for the description or construction of the different viewpoint models. These should be adequate for the specific purpose of the viewpoint model, both concerning the addressed aspects and the envisaged users. This approach is in principle also supported by the Unified Modeling Language [20] that provides different modelling languages for the specification

* This work has been supported by the research project IOSIP (Eh65/10-2) within the DFG Priority Program *Integration of Software Specification Techniques for Applications in Engineering* (1064)

of the static structure, the dynamic behaviour (from different points of view), and the implementation and deployment of the system.

But even if the viewpoint model is not explicitly referred to in the development process, one always has to deal with heterogeneous models and languages. In the development of embedded systems in an engineering application for example one has to communicate with engineers who use their established languages for the description of the technical (and also software technical) part of the system ([1]). In an integration of different information sources in a federated information system different specifications and specification formalisms have to be considered ([10]), and when systems are used as communication infrastructures continuously over a long period new models and specifications, given perhaps in new languages, must be related to older ones ([2, 11]).

Thus, the use of different languages cannot and should not be avoided, but mechanisms for their integration should be provided. In the research project IOSIP[1] a reference model for the integration of heterogeneous software specifications has been developed that supports a formal semantic approach (see [5, 7, 8]). A global semantic domain is given where all specification languages can be directly and adequately interpreted and compared. This allows the investigation and solution of semantic problems independently of concrete specification paradigms and languages, before they are represented syntactically w.r.t. the considered concrete languages.

The elements of the semantic domain are *transformation systems* that represent the static structure and dynamic behaviour of arbitrary entities (systems, components, objects, processes, methods, etc.). Roughly, these are extended labelled transition systems where both the states and the transitions are labelled. (Using graphs as state labels one obtains thus graph transformation systems as a special case of transformation systems.) The structure of the domain of transformation systems is given by general schemes for *development relations* that express semantic compatibility properties like refinement and implementation for example, and *composition operations* for the construction of larger entities by the interconnection and synchronization of smaller ones.

The domain of transformation systems yields a semantic background for the analysis and integration of specifications. Unlike other approaches that are based on the transformation of structures (like graph transformations) it is not intended as a new specification or modelling language, although it could also be used for this purpose, too.

In the remainder of this paper we briefly sketch the semantic domain of transformation systems, describe different representations of static structures that arise in different application contexts and require more expressive means than graphs, and discuss some applications of the reference model induced by the transformation systems approach.

[1] URL http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/iosip.html

2 Transformation Systems

A transformation system is a transition system (i.e., a graph) with labels for the states and the transition. The state labels represent the *data states* of the modelled entity. They are given by the class of all instances of a schema or signature that defines the static structure of the entity. The state of an object for example is given by the values of its attributes. In this case the signature is given by the list of the attributes of the object and their types, and a state is given by a corresponding list of values, i.e., elements of the types. Likewise, a program's signature and its state are given by the list of (typed) program variables and their values respectively. In graph transformation system approaches various kinds of graphs are used to represent data states. But data states can be arbitrarily complex and require further means for their representation, as discussed in the following section. Therefore the definition of transformation systems is parameterized w.r.t. the kind of structures used for the representation of data states. (Formally the framework for the data state specification parameter is required to be a concrete institution, see [8].) Analogously, transitions can be labelled by atomic actions, parameterized actions, sets of (parameterized) actions for the representation of concurrent systems, or arbitrary other *action structures*.

According to this two level structure of transformation systems—the transition graph representing the skeleton of the dynamic behaviour and the data states and action structures for the static structure—their properties can be classified. *Data invariants* are properties that are satisfied in each data state of the system. They can be expressed in the language induced by the data states signature. A single step $T : C \Rightarrow D$ of a transformation system is given by one transition (edge) $t : c \to d$ of the transition graph and its labels, the data states C and D associated to c and d and the action structure T associated to the transition t. The step $T : C \Rightarrow D$ represents a data state transformation induced (caused, effected) by the actions T. Correspondingly, *transformation properties* can be specified by *transformation rules* $\alpha : L \Rightarrow R$, given a pair of individual data state specifications L and R for C and D respectively and an expression α for the action structure T. Note that these transformation rules are not considered constructively as replacement rules, but descriptively like pre and post conditions. (However, a constructive interpretation can also be given, provided the data state specification formalism is algebraic, see [6].) Finally *control flow properties* refer to the whole transformation system, i.e., its transition graph and the labels. In contrast to the data state and transformation specifications there is no canonical format for the control flow specification. Any formalisms whose (operational) semantics can be defined in terms of (extended) labelled transition systems can be used. Examples are process calculi like CCS [13] and CSP [9], and Petri nets [15] for constructive specifications and temporal or modal logics [4] for descriptive ones.

The structure of the domain of transformation systems is given by *development relations* that support the semantic comparison of systems and *composition operations* to construct larger systems from smaller ones. Both follow in their definition the above mentioned two level structure of transformation systems.

A development relation of two transformation systems is given by a graph homomorphism and a signature morphism for the relation of their transition graphs and data state and action signatures respectively. The former states how the behaviour of one system can be embedded into or simulated by the other one. With different kinds of graph homomorphisms additional behaviour (extended functionality) or restriction (exclusion of certain actions, reduction of non–determinism, etc.) can be represented. The signature morphism states how the structure of one systems is embedded into or simulated or refined by the other one. It induces an operation (a forgetful functor) on the data states and action structures that realizes the desired relation, restricting for example the set of visible attributes and actions. The two morphisms can be combined either in the same direction or in opposite directions. Moreover, closure operations can be used that allow the mapping of a single step of one system to a (sequential, parallel, etc.) composition of steps of the other one. In this way a schema for the definition of different kinds of development relations is given. It can be adapted and instantiated to meet both the development relations supported by the considered specification languages and the transformations that are necessary to compare and integrate different viewpoint specifications.

Analogously a schema for the definition of composition operations is given. In general, transformation systems are composed by specifying how their static structures are related and how the systems shall be synchronized. The former information is given by an *identification relation*, i.e., a relation on the signatures of the systems, that states which parts of their static structures are shared (pervasive static data types likes numbers and strings, exchange data types used in the communication, or shared variables or other shared structures used to realize communications for example.) The synchronization is specified by a *synchronization relation* on the transition graphs of the systems that declares which states and transitions of the systems are compatible with each other in the sense that they can be entered or executed simultaneously. That means, states or transitions may be synchronized if they are in the synchronization relation. Must–synchronization can be obtained as a special case. The result of the composition of transformation systems connected in this way represents a global view of the interconnected local systems as a single transformation system again. It is given by all tuples of synchronous states and transitions respectively, whose labels are given by the *amalgamations* of the local data states and action structures respectively. The general schema for the composition operations can be instantiated again to reconstruct the composition operations supported by the considered specification languages and to (semantically) express the compositions that are necessary for example to compare and integrate specifications of different scopes.

The definition of the composition operations supports structural transparency in that it allows us to abstract from the internal architecture of a composed system and consider it as a single system with one global static structure and one global behaviour. According to the formal definition of the properties, development relations and composition operations moreover general compositionality properties have been shown that state under which conditions properties of the local components are preserved by a composition and how compatible local developments induce a development of the global system.

3 Static Structure Representations

As mentioned above transformation systems are generic w.r.t. the representation of data states and action structures. That means, when using the approach an (almost) arbitrary framework (institution) for the specification of the data states and actions can be employed. Lists of typed attributes of objects and typed program variables have been mentioned above as most simple examples. Different kinds of graphs, like labelled, typed and/or attributed graphs or graph like structures are used in graph transformation based approaches like AGG [19], PROGRESS [16, 17] and others [3]. In the *standard version* of transformation systems *partial algebras* are used as data state models. This allows us to include and reason about static data types explicitly, to represent parameterized attributes as partial functions, and to use mutable sets, for example to represent the varying set of references an object maintains or to represent creation and deletion of items generally. Obviously, all kinds of graphs mentioned above are special cases of partial algebras. Furthermore states of (elementary, coloured, algebraic, etc.) Petri nets can be represented in different ways as partial algebras.

In some applications, however, even partial algebras are not sufficient to represent data states adequately, i.e., without encodings. Especially when larger entities like software components or whole systems are considered further means are required.

Consider for example a class diagram that is used for the specification of the static structure of a system. In order to describe a state of the system the following information has to be given.

- How many objects of each class are there and how are the objects linked?
- What is the state of each object?

A system state (as data state) can accordingly be described as follows (see Figure 1).

- A graph defines the objects via its nodes and their links via its edges. Considering the class diagram as a graph, too, with classes as nodes and associations as edges, the typing of the objects and links can be defined as a graph homomorphism from the object graph to the class graph.
- To define the objects states first each class in the class diagram is considered as an algebraic signature as follows. The types that occur in the class as attribute or parameter types yields sorts, possibly with associated function symbols. Types may be classes or built–in static data types. The attributes are represented as constants in the signature. Moreover, for each operation op of the class with return type t a constant $ret\_op : t$ is introduced that serves to hold the return values of the operation in the states after its execution. Partial functions can be used, moreover, if parameterized attributes

are considered for example. The partial algebras of this signature then represent the possible data states of the objects of this class. They contain the static data types and values for the attributes and the return values of the operations (which may also be undefined). The carrier sets corresponding to the class types contain the references an object maintains internally to refer to other objects.
To access the other objects in addition corresponding dereferencing functions have to be given. The domain of a dereferencing function is thereby given by a carrier set of a class sort (i.e., a set of references inside an object), its codomain is given by the set of objects of the corresponding class, i.e., the set of those nodes of the object graph that are mapped to the class node in the class graph.

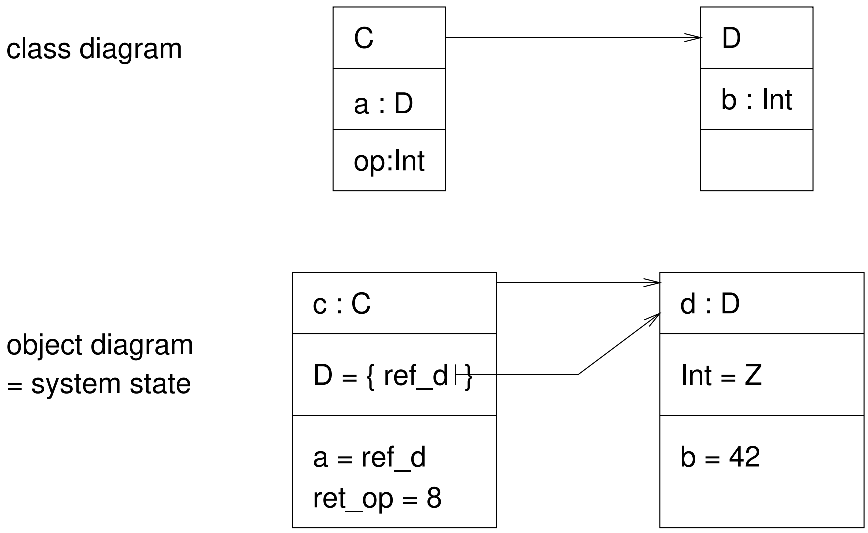


Fig. 1. Data state of a system, given by the objects, their links, their data states, and the dereferencing

The dereferencing functions cross the frontiers between the objects insides and the entire system. This cannot be expressed, for example, with attributed graphs, since there only functions from the graph into the label algebra are allowed, whereas the dereferencing function have opposite functionality. We thus have a direct and adequate representation of system states that formally fits into the framework of transformation systems. Moreover it supports the navigation along links and references and the specification of state properties like OCL data invariants.

The behaviour of single objects is specified in the UML with state machines, that also refer to the static structure via actions and conditions. Thus the static aspects have to be interpreted, too. The semantics of state machines can also be defined in terms of transformation systems. Thereby data states are represented similarly as system states in the interpretation of class diagrams discussed above. A restriction is thereby given in that only a single object is considered. However, this object must be seen in its context, because it may refer to other objects in the system. Accordingly, the data state of an object is defined now as a *partial system state* like above, where, however, the data state (partial algebra) and the dereferencing functions are given only for the considered object. That means, the object may refer to other objects but may not inspect their states. The data state of a state machine is then given by the data state of the object, i.e., an instance of the class the state machine is associated to, and a configuration of the state machine. The latter is given by a compatible set of basic states of the state machine, one within each parallel region of the machine, and represents the reactive capability of the object as far as specified within the state machine. Note that the conditions that guard the transition are not interpreted in the state machine itself. Therefore, in the semantics, the data states of the objects have to be included, too.

The whole information represented in a data state of an object specified by a state machine (that implicitly refers to a class diagram in addition) is needed to define the semantics of single objects. Moreover, the *interactions* of sets of objects of different classes are already determined by their state machines, too. Indeed, they are completely specified by the event–action duality that yields the synchronizations of the objects. The semantics of these interactions can also be defined formally using the composition operations of transformation systems. Thereby the identification relation only identifies the common static data types, and the synchronization relation is induced—as mentioned above—by the duality of events and actions. (An action of one object is an event for another one.)

4 Applications

The main aim in the development of transformation systems has been to provide a reference model for the integration of heterogeneous formal and semi–formal software specifications. Integration is thereby interpreted conceptually as the possibility to consider a collection of (heterogeneous) viewpoint specification as a single specification of one system. This does not necessarily mean to syntactically integrate specifications, but only yields a background that has to be adapted and instantiated to obtain a concrete applicable integration method. Nevertheless, the semantic foundation have been worked out in detail and now support the development of integration methods in a precise and clear way. (That means, the old slogan *Semantics First!* has been respected.)

The meaning of integration can now be described more precisely as follows. First of all, each specification in a development according in one way or another to the viewpoint model is a partial, incomplete specification, since it only specifies one aspect of a system. W.r.t. the other aspects that are not addressed in this viewpoint the system is not constrained by the considered viewpoint specification. That means, semantically, each specification admits a *set of interpretations*. When a collection of specifications is to be integrated basically the intersection of these sets of admissible interpretations yields the conjunction of the specifications on the semantic level. However, some transformations might be required before that, because the scopes of the specifications, i.e., that part of the system they specify, and their granularity w.r.t. structure and behaviour may be different. For example, a class diagram specifies arbitrary collections of objects, a statechart diagram specifies a single object, and a collaboration diagram specifies a definite set of objects. Thus appropriate projections have to be applied to the semantic interpretations of the individual specifications. On the other hand, the transformation systems representing single object behaviours can also be composed in order to obtain system models that can be compared—by intersection—with interpretations of collaboration and class diagrams for example. The conceptual semantic integration can be summarized thus as *the intersection of transformed sets of admissible interpretations*. The transformations thereby represent semantic correspondences of the considered specifications. Based on these the intersection reflects their consistency. If the intersection is empty the specifications are inconsistent (w.r.t. the correspondences expressed by the transformations), if it has more than one element the set of viewpoint specifications still admits different interpretations, i.e., it can be considered as incomplete.

Based on this abstract integration concept concrete integration methods can be developed. As opposed to the general language and paradigm independent approach provided by the reference model an integration method should be developed for concrete languages or at least specific classes of languages. Then the roles of the concerned viewpoints and the recommended usage of the languages can be taken into account and reflected in the method. For example, class diagrams only define the static structure of systems and thereby provide the syntactic means that are used in the other specifications. (In this sense they are a precondition for the development of the other models.) A class diagram induces the data states and action structures of the transformation systems of its admissible interpretations, but it does not constrain their transition graphs. The order of the invocations of the operations provided by the classes is not given. Thus, after the interpretation of the class diagram the other diagrams are used to select those transition graphs that represent admissible interpretations of the behaviours specified by a state machine or a collaboration.

Beyond the integration of individual specifications the reference model also yields guidelines how to integrate specification languages. The parametric definition of transformation systems indeed states how a specification framework for data states (= an abstract data type specification method) and a control flow specification framework (= behaviour or process specification method) are integrated to obtain the corresponding instance of transformation systems. The abstract data type specification method induces a data transformation specification framework, since a transformation is essentially given by pairs of data states, augmented by an action structure (the latter has to be given in addition). The behaviour specification is then put on top of the framework and used to define the order (temporal, causal, etc.) of the steps defined in the data transformation specification framework. The behaviour specification may thereby refer to both the action structures and the data states of the lower level, which justifies the hierarchical integration of data state and behaviour specification.

When the transformation system reference model is used for the integration of concrete specification languages theses have to be interpreted in terms of transformation systems. That means, their semantics has to be reconstructed or defined by transformation systems, development relations, and composition operations as provided by the reference model. (A series of examples for such interpretations is given in [8], where also the adequacy and sufficient expressivity of the reference model is discussed in more detail.) Thereby the explicit structures and the properties of the reference model can be reflected to the language. That means, the interpretation can be used to suggest semantics definitions for aspects that have not yet been defined precisely and to introduce development operations or composition operations with precise semantics. The interpretation of the UML techniques discussed above for example also provides a precise definition of the object interaction specified by state machines, which has not been addressed yet in the semantics definition at all. Moreover, the reference model provides development relations that can be used to define the semantics if subtyping and inheritance w.r.t. the state machines associated to the corresponding classes for example.

Using the constructions employed for the definition of the composition of transformation systems, basically identification and synchronization relations for the interconnection of local systems and their representations, then also different *architecture description languages* can be analyzed, compared, and possibly integrated (see [12, 18]). This aspect, however, has not yet been investigated in detail.

References

1. A. Braatz, M. Große–Rhode, H. Ehrig, and E. Westkämper. UML-basierte Software-Spezifikation und Entwicklungswerkzeuge für Systeme der Automatisierungstechnik. In *Proc. Engineering komplexer Automatisierungssysteme*, April 2001.
2. J. Cheesman and J. Daniels. *UML Components: A simple process for specifying component-based software.* Addison–Wesley, October 2000.
3. H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages and Tools.* World Scientific, 1999.
4. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 16, pages 995–1072. Elsevier and MIT Press, 1990.
5. M. Große–Rhode. A compositional comparison of specifications of the alternating bit protocol in CCS and UNITY based on algebra transformation systems. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. of the 1st International Conference on Integrated Formal Methods (IFM'99), York, UK, 28–29 June 1999*, pages 253–272. Springer Verlag, 1999.

6. M. Große–Rhode. Algebra transformation systems as a unifying framework. In M. Bauderon and A. Corradini, editors, *Proc. GETGRATS Closing Workshop*, volume 51 of *Electronic Notes in Theoretical Computer Sciences*. Elsevier Science Publishers, 2001.
7. M. Große–Rhode. Integrating semantics for object-oriented system models. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Proc. International Colloquium on Automata, Languages and Programming (ICALP 2001)*, pages 40–60. Springer LNCS 2076, 2001.
8. M. Große–Rhode. Semantic integration of heterogeneous formal specifications via transformation systems. Technical Report 2001/13, TU Berlin, August 2001.
9. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
10. R.-D. Kutsche, S. Conrad, and W. Hasselbrink, editors. *Engineering Federated Information Systems, Proc. 4th Workshop EFIS, October 9–10, 2001, Berlin, Germany*. Akademische Verlagsgesellschaft Aka, Berlin, 2001.
11. S. Mann, B. Borusan, H. Ehrig, M. Große-Rhode, R. Mackenthun, A. Sünbül, and H. Weber. Towards a component concept for continuous engineering. Internal Report of the BMBF Research Project *Continuous Engineering*, April 2000.
12. N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
13. R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
14. ISO/IEC International Standard 10746, ITU–T recommendation X.901–X.904: Reference model of open distributed processing – Parts 1–4.
15. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.
16. A. Schürr. PROGRES: A vhl-language based on graph grammars. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*. Springer, 1991.
17. A. Schürr, A. Winter, and A. Zündorf. The PROGRES-approach: Language and environment. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.
18. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice–Hall, 1996.
19. G. Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99), LNCS 1779*, pages 481–490, 2000.
20. *Unified Modeling Language – version 1.3*, 2000. Available at http://www.omg.org/uml.

Automated Program Generation *for* and *by* Model Transformation Systems*

Dániel Varró

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems
H-1521 Budapest Magyar tudósok körútja 2.
varro@mit.bme.hu

Abstract. Model transformation systems are graph transformation systems that perform translations between languages defined by a corresponding metamodel as the type graph. The current paper proposes a reflective method for the automatic generation of the implementation for such transformation systems derived from a high–level specification consisting of a set of graph transformation rules and a control flow graph. The program generator takes a UML profile tailored to model transformation systems as the input, and produces the output Prolog program by successive model transformation steps. In this respect, only the core of the program generator is implemented by hand, and afterwards, this core provides automation for additional features of the VIATRA model transformation system.
Keywords: model transformation, graph transformation systems, automated program generation

1 Introduction

1.1 Model transformations in system design

Although the Unified Modeling Language (UML) has become the de facto standard visual modeling language of object–oriented design, both academic investigations and engineering experiments have revealed several shortcomings regarding, especially, its imprecise semantics and the lack of flexibility in domain specific applications [9]. Recently, the UML 2.0 Request For Proposal issued by the OMG has addressed to re-architecture the single and monolith language into a family of languages with individually defined semantics based on a kernel metamodeling language.

However, as the formal semantics of different views of the system (i.e., separate diagrams like class diagrams, statecharts, sequence diagrams, etc.) might be defined in different semantic notations (e.g., by Petri nets, SOS rules, graph transformation systems etc.), the integration of such local views into a consistent global view of the system requires a precise specification of transformations *within* and *between* UML models.

* This work was supported by the Hungarian National Scientific Foundation Grant (OTKA 030804)

In practice, transformations are necessitated for several purposes: (i) *model transformations within a language* should control the correctness of consecutive refinement steps, (ii) *model transformations between different languages* should provide precise means to project the semantic content of a diagram into another one, which is indispensable for a consistent global view of the system under design, and (iii) a visual UML model (i.e., a sentence of a language in the UML family) should be transformed into its semantic domain, which process is called *model interpretation*.

As the abstract syntax of UML models is defined visually by a corresponding metamodel, a straightforward representation of such models can rely on the use of directed, typed, and attributed graphs for the underlying semantic domain. Therefore, the use of graph transformation for capturing the semantics of model transformations is a natural choice which also fits well to engineering practices as a consequence of its visual expressiveness [20].

However, even if the formal specification of a transformation is precise (and formally verified), *its implementation is still highly error prone due to a huge abstraction gap between visual UML models and formal mathematical descriptions.* For this reason, the automated generation of a program that implements the transformation is also a major requirement for model transformation systems.

1.2 VIATRA: Visual Automated Model Transformations

VIATRA (VIsual Automated model TRAnsformations) is a prototype tool being developed at the Budapest University of Technology and Economics, that provides a general means to specify and implement various transformations between models defined by their corresponding metamodels by the paradigm of graph transformation.

VIATRA interface The user interface of VIATRA is a set of XMI [1] files, which includes descriptions of metamodels, models and their transformations in all phases of model transformations.

1. **Metamodels:** Both UML and mathematical languages are defined by a corresponding metamodel (e.g., a metamodel for Petri nets, or state transition systems, etc.), which is constructed in a commercial UML CASE tool having XMI export facilities.
2. **Transformations:** The elementary model transformation steps are defined by specially structured graph transformation rules while the entire computation is specified by a control flow graph. In practice, both graph transformation rules and control flow graphs are described in a high-level UML notation using a transformation specific profile based on UML Class diagrams.
3. **Models:** For obtaining a flexible and general interface, the input and output models of VIATRA (thus both UML and mathematical models) are XMI files conforming to their metamodel.

[1] XMI (XML Metadata Interchange) is the standard XML-based description format for systems based on MOF metamodeling.

VIATRA features The current version of VIATRA supports the following (main) features.

1. **Automated DTD generation:** A corresponding DTD (Document Type Definition) can be generated automatically from metamodels.
2. **Automated program generation:** When VIATRA is supplied with the input files of the transformation and the metamodels, it automatically generates the declaration of the transformation (in Prolog) including the implementation of the control flow graph and graph transformation rules.
3. **Automated transformation:** This transformation program is then executed on an arbitrary source model, and the target model is generated. Although VIATRA *currently* uses Prolog as the underlying transformation engine (i.e., the concrete transformation of models is performed in Prolog), it is hidden from the user and it still provides a more efficient solution than by using an XSLT [2] engine.

The current paper provides *an overview of the underlying program generation method* of VIATRA (a more detailed description can be found in [19]). The main characteristics of our solution are (i) the use of *intermediate transformation steps* to avoid re-implementing code generation for executable programs and input descriptions for model checking tools, (ii) the *reflective specification method* embedded in the code generation process (the implementation of model transformation systems is specified by model transformation systems), (iii) and a (future) *bootstrapping* step to improve the quality of the code generation (when a previous version of the program generator is used for generating the next version of the program generator, in analogy with the well–known bootstrapping techniques of compiler design).

2 Automated Program Generation for Model Transformation Systems

2.1 Theoretical background of model transformations

We provide a brief overview of the main concepts of model transformations, namely, model graphs, model transformation rules and model transition systems.

Definition 2. *A* ***model graph*** *G is a directed, typed and attributed graph. The type graph of a model graph is called the* ***metamodel****, which is related to a model graph by a typing homomorphism. The* ***metametamodel*** *is the common language (in other words, the top-most type graph) for describing metamodels, which is reflective (i.e., its type graph is itself).*

In model transformation systems, the source and target models are related by a reference graph, which is, in fact, an ordinary model graph. In general, a reference graph is a common abstraction of the source and target models relating the corresponding nodes and edges to each other.

[2] XSTL (eXtensible Stylesheet Language Transformation) is an XML technology used for describing (mainly syntactic) transformations between XML files.

Definition 3. *A* ***model transformation rule*** $r = (L, N, R, M)$ *is a special graph transformation rule, where all graphs* L*,* N *and* R *are model graphs applied in the specified mode* M*.*

The **application** of r to a **host graph** G replaces an occurrence of L (left-hand side) in G by an image of R (right-hand side) yielding the derived graph H. This is performed by

1. *finding an occurrence* of L in G, which is either an isomorphic or non–isomorphic image according to M
2. *checking the negative application condition* N, which prohibit the presence of certain nodes and edges
3. *removing* those nodes and edges of the graph G that are present in L but not in R yielding the **context graph** D (all dangling edges are removed at this point)
4. *adding* those nodes and edges of the graph G that are present in R but not in L attaining the **derived** graph H.

Currently, the behavior of VIATRA follows conceptually the single pushout approach [6] (i.e., removing dangling edges and allowing non–isomorphic images in graph pattern matching), however, the concrete graph manipulations are defined and implemented by logics based rewriting showing closer correspondence to the techniques of [16].

The entire model transformation process is defined by an initial graph manipulated by a set of model transformation rules (micro steps) executed in a specific mode in accordance with the semantics (macro steps) of a hierarchical control flow graph.

Definition 4. *A* ***model transition system*** $MTS = (Init, R, CFG)$ *with respect to (one or more) type graph* TG *is a triple, where* $Init$ *defines the* ***initial graph****,* R *is a set of* ***model transformation rules*** *(both compatible with* TG*), and* CFG *is a set of a* ***control flow graphs*** *defined as follows.*

- *There are six types of nodes of the CFG, each associated with a rule* $r \in R$*:* ***Start****,* ***End****,* ***Try****,* ***Forall****,* ***Loop*** *and* ***Call****.*
- *There are two types of edges:* ***succeed*** *and* ***fail****.*

The control flow graph is evaluated by a virtual machine which traverses the graph according to the edges and applies the rules associated to each node.

1. The execution starts in the **Start** and finishes in the **End** node. Neither types of nodes have rules associated to them.
2. When a **Try** node is reached, its corresponding rule is tried to be executed. If the rule was applied successfully then the next node is determined by the **succeed** edge, while in case the execution failed, the **fail** edge is followed.
3. At a **Loop** node, the associated rule is applied as long as possible (which may cause non-termination in the macro step).

4. When a **Forall** node is reached, the related rule is executed parallelly for all distinct (possible none) occurrences in the current host graph.
5. Finally, at a **Call** node (which has an associated CFG and not a rule) the state of the CFG virtual machine is saved and the execution of the associated CFG is started (in analogy with function calls in programming languages). When the sub CFG machine is terminated, the saved state is restored, and the execution is continued in accordance with the outgoing edge (**succeed** or **fail**).

2.2 A case study: Semantics of Message Sequence Charts

To provide a more deeper insight into the expressiveness of model transformation systems, below we consider a semantic interpretation of Message Sequence Charts (abbreviated as MSCs in the sequel). The semantics of MSCs that define a partial order on events (following the semi-formal description in [14]) is captured by a corresponding model transformation system.

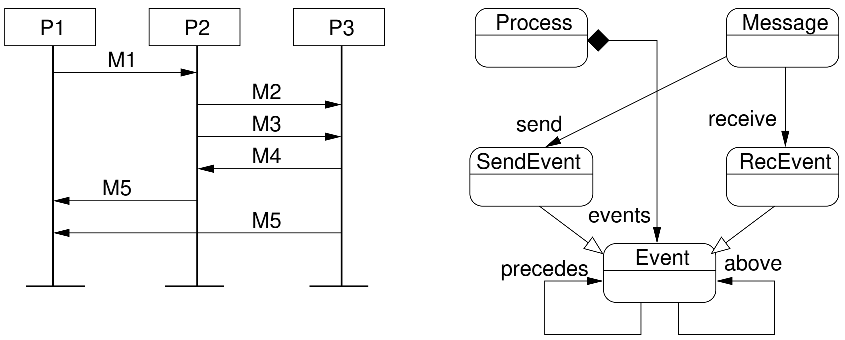


Fig. 1. Message Sequence Charts: visual syntax and metamodel

Message Sequence Charts (a sample MSC model is depicted in Fig. 1 together with a metamodel) are constructed from **Processes** (depicted as *rectangles*) that communicate by sending and receiving **Messages** (shown as *arrows*). The fact that a message is sent or received is represented by a corresponding **Event** (**SendEvent** or **RecEvent**) on the process line (depicted as *vertical lines*). If a message M_i is (supposed to be) sent before another message M_j then the arrow representing M_i appears above the arrow of M_j.

For the semantic interpretation, we define a partial order on MSC events and formalize it by the model transformation system of Fig. 2 as follows.

- **Causality.** If p is the send event and q is the receive event of the same message then p precedes q.
- **Controlability.** If p appears above q on the same process line, and q is a send event then p precedes q. This order reflects the fact that a send event

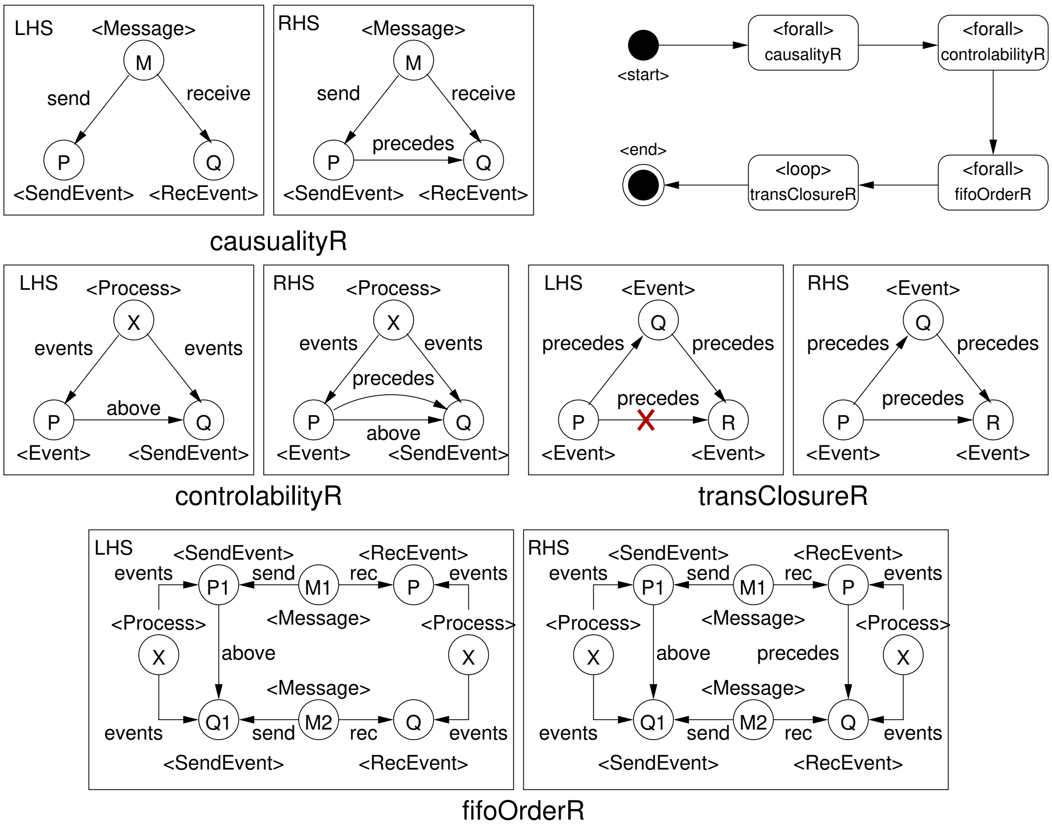


Fig. 2. Defining semantics for Message Sequence Charts

can wait for other events to occur. On the other hand, we typically have less control on the order in which receive events occur.

– **FIFO order.** For any send events p' and q' on the same process line where p' is above q', p precedes q for the corresponding receive events p and q.
– **Transitivity.** The *precedes* relation is transitive, i.e., if p precedes q and q precedes r then p precedes r.

Example 6. We select rule *transClosureR* for deeper investigations. According to the control flow graph, this rule is applied in *loop* mode in the very end of the semantic transformation process, and generates the transitive closure of the *precedes* relation. If event `P` is already connected to `Q` by a `precedes` edge and `Q` is already connected to `R` (according to the left-hand side of the rule), and if `P` is not connected yet to `R` by a `precedes` edge (negative condition), then a new `precedes` edge is added leading from `P` to `R` (as defined by the right-hand side). *Loop* mode prescribes that rule *transClosureR* should be applied as long as possible.

The rules of the corresponding model transformation system are rather straightforward with respect to the metamodel and the informal semantics of MSCs, which clearly demonstrates the descriptive power of our approach. Even though the control flow graph of our case study is relatively simple, the use of CFGs for restricting the valid computations still helps reduce the complexity of rule preconditions (thus resulting in more efficient pattern matching). In our case, we can safely omit the negative application conditions from rules that can be executed in *forall* mode (such as *causalityR*, *controlabilityR* and *fifoOrderR*) without duplicating *precedes* edges since those rules are applied parallelly in a single (deterministic) transformation step.

2.3 Program generation for graph transformation rules

The automated program generation of VIATRA allows the transformation designers to focus on the *design* of a model transformation rather than the *implementation*. Previous experiments (in project HIDE [3]) demonstrated that the quality of an automatically generated executable transformation program is much higher than a manually written target program. Moreover, once the automated program generator is completed, the time and workload related to the design of a single transformation is drastically decreased.

In VIATRA, the program generation process of model transformation rules is divided into several intermediate steps.

1. Model transition systems are specified in a UML CASE tool (Rational Rose was used for our experiments), and exported in the standard XMI format.
2. This UML model is transformed into a GraTra model conforming to a metamodel of graph transformation systems.
3. In VIATRA, model graphs are represented as predicates in a fact database. For this reason, the previous GraTra model is projected into a Logic model containing a sequence of terms for each rule.
4. The bridge between visual (graph based) and the textual language of Prolog is provided by the parse tree of the Prolog code. In this sense, the Logic model is transformed into a corresponding Prolog parse tree, and the target Prolog code is printed out by traversing this tree in an in-order way.

The importance of these intermediate steps is threefold.

– At first, there is a huge abstraction gap between a visual UML-based specification of the transformation and even such a high-level programming language as Prolog. Thus splitting the transformation into several subtransformations decreases the complexity of the individual steps, which eases not only the implementation but also the verification of the automated program generation.
– Secondly, the use of intermediate models increases reusability. For instance, when generating the input language of a model checker for the verification of a model transformation (an ongoing research activity), only the final step needs to be altered.

- Finally, the intermediate GraTra model provides the right basis for generating the upcoming standard XML description of graph transformation systems [18] by a simple transformation from the GraTra XMI format to GXL/GTXL.

In fact, each intermediate transformation step is specified (and implemented) as model transformation, which means that *the entire code generation process is captured by graph transformation*. In this sense, the implementation of model transformation rules is specified by model transformation rules. This approach is similar to the bootstrapping process of compiler design, where, for instance, a C compiler is written in C and compiled by an existing C compiler, and recompiled by itself afterwards to provide a more efficient and reliable target code. In VIATRA, the current version of the program generator is implemented manually, while future versions (with additional features, and more efficient / reliable target code) are generated by using the existing version of the program generator.

```
causalityR:-
  node(msc:message(M)),
  edge(msc:send(E1,M,P)),
  node(msc:sendEvent(P)),
  edge(msc:receive(E2,M,Q)),
  node(msc:recEvent(Q)),
  add(edge(msc:precedes(E3,P,Q))).
```

```
controlabilityR:-
  node(msc:process(X)),
  edge(msc:events(E1,X,P)),
  node(msc:event(P)),
  edge(msc:events(E2,X,Q)),
  node(msc:recEvent(Q)),
  edge(msc:above(E3,P,Q)),
  add(edge(msc:precedes(E4,P,Q))).
```

```
fifoOrderR:-
  node(msc:process(X)),
  edge(msc:events(E1,X,P1)),
  node(msc:sendEvent(P1)),
  edge(msc:events(E2,X,Q1)),
  node(msc:sendEvent(Q1)),
  edge(msc:above(E3,P1,Q1)),
  edge(msc:send(E4,M1,P1)),
  node(msc:message(M1)),
  edge(msc:receive(E5,M1,P)),
  node(msc:recEvent(P)),
  edge(msc:send(E6,M2,Q1)),
  node(msc:message(M2)),
  edge(msc:receive(E7,M,Q)),
  node(msc:recEvent(Q)),
  add(edge(msc:precedes(E8,P,Q))).
```

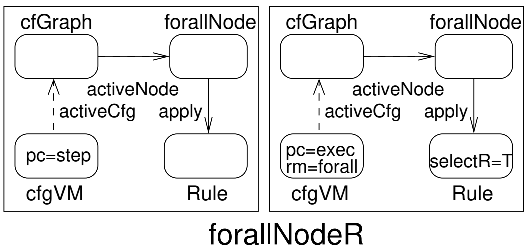
```
transClosureR:-
  node(msc:event(P)),
  edge(msc:precedes(E1,P,Q)),
  node(msc:event(Q)),
  edge(msc:precedes(E2,Q,R)),
  node(msc:state(R)),
  ( edge(msc:precedes(E3,P,R) ->
    fail ; true),
  add(edge(msc:precedes(E4,P,R))).


msc:-
  forall(causabilityR),
  forall(controlabilitR),
  forall(fifoOrderR),
  loop(transClosureR).
```

Fig. 3. Program generation for transformation rules

Example 7. We continue our case study with a brief insight into the structure of the generated Prolog code (see Fig. 3). Again, we discuss only the behavior of rule *transClosureR* in details. The Prolog code of the rules implements the graph pattern matching by consecutive unifications during which the variables `P, Q, R, E1, E2` are instantiated. The outermost terms (`node` and `edge`) are responsible, for instance, for the hierarchical matching of patterns (i.e., a node of type *SendEvent* should also be matched by the *Event* pattern in case of MSCs). The negative part (within parenthesis) causes failure for the current matching if and only if `R` is already a substate of `P`, and then steps to the next matching to be examined by backtracking. Finally, after a successful pattern matching, a `precedes` edge is added between `P` and `R`.

The example also demonstrates that the generated program partially contains transformation dependent (translated) Prolog code (i.e., the sequence of terms representing queries on the model graph) and (interpreted) calls to built-in routines from a VIATRA library (like `node()`, `edge()`, and routines implementing different modes of rule application discussed in the upcoming section).

2.4 Implementing the virtual machine




```
forall(Rule):- call(Rule),  fail.
forall(Rule).
```

Fig. 4. Executing a step of the virtual machine

The implementation of the virtual machine that executes the control flow graph (CFG) of a model transition system builds upon the reflective property of program generation in VIATRA as the operational semantics of this virtual machine is also defined by model transition systems. In this respect, the program corresponding to a rule is generated automatically, and the implementation of the "meta" CFG (i.e., the CFG that defines the behavior of the virtual machine) is very simple. The entire semantics of the virtual machine consists of 11 rules, from which the handling of *forall* nodes is depicted in Fig. 4.

Example 8. The semantics of rule *forallNodeR* is as follows. When the virtual machine is to execute a step (`pc = step`), then the active CFG node in the active graph should be found and the `selectR` attribute of the associated graph transformation rule is set to true in order to select the rule for execution. Moreover, the program counter `pc` is set to `exec` indicating that now the virtual machine should execute the rule before making the next step and attribute `rm` (which represents the execution mode of a rule) is set to `forall`.

In the rule execution phase, the piece of code in the right region of Fig. 4 is called, which is responsible for executing the rule for all possible matches in the current state. This is obtained by causing an artificial backtracking whenever the rule application is succeeded. Finally, if all possible matches are processed, the application of a rule in *forall* mode is also successfully completed (including the case when there are no possible matches of the LHS).

Finally, the current node of the control flow graph is updated according to the success of rule application, and we proceed with the new current node.

3 Conclusion

In the current paper, we presented an automatic program generation framework *for* the implementation of model transformation systems, where the process of program generation is specified *by* consecutive model transformations. Our approach (implemented in the VIATRA tool) is reflective in the sense that the next versions of the program generator can be derived by the previous version similarly to the bootstrapping techniques of compiler design. In fact, the virtual machine of VIATRA that executes a control flow graph has been implemented by using the automated program generator of graph transformation rules.

VIATRA has already been applied successfully to provide an automated implementation of transformations specified by means of model transition systems.

- The specification (and implementation) of the VIATRA virtual machine required 11 graph transformation rules and a simple automata.
- In [11], a transformation from UML Statecharts to Extended Hierarchical Automata (EHA) has been carried out that provides formal operational semantics for statecharts. In [19], we formalized the entire transformation by model transitions systems with over 40 rules.
- The original paper defined the EHA semantics as a Kripke automata. We also provided visual semantics for EHA by model transition systems having approximately 20 rules (with simple LHS graphs).
- The completeness of UML statechart specifications in a dependable environment has been investigated in [13]. Currently, an automated verification program is under implementation using VIATRA.
- A transformation from UML statecharts to Stochastic Reward Nets [4] is also under implementation (having currently 25 rules for a well–separated subproblem) to provide access to Petri Net based analysis techniques.

3.1 VIATRA as a graph transformation tool

As state-of-the-art tools of graph transformation systems (such as GenGEd (with AGG) [1] , DiaGen [10], Progres [17], and FUJABA [12]) have been evolving for more than a decade, VIATRA is naturally not the only tool that is capable of performing model transformations. However, we believe that VIATRA is tailored to the special needs of model transformations between UML and semantic domains to such an extent that makes our tool more flexible and powerful in this specific application domain of graph transformation systems than sophisticated general-purpose graph transformation tools. Therefore, it is rather (i) the underlying model transformation methodology, (ii) its openness and compliance with leading industrial standards, and (iii) its (ongoing) integration with model checking tools that makes VIATRA unique rather than the core graph transformation engine itself.

- **Openness, Compliance with standards:** VIATRA is an open environment built around XMI technology, which is the de facto standard in UML-based modeling environments. XMI DTDs for non-UML models are generated automatically from metamodels, which is a more flexible solution in domain specific applications than tools forcing to use a fixed set of XML elements. Moreover, similarly to the story diagram-based rule descriptions [8] of FUJABA, VIATRA uses a UML profile based on class diagrams as the formal specification language of model transformations.
- **Model transformations:** Transformations of UML models necessitate to manipulate complex data structures with *a large number of rules* (see our benchmark transformations or [7], where a Java implementation of UML models is also specified by graph transformation rules), which makes *graph transformation tools without control condition impractical* for such applications due to the increased level of nondeterminism. In addition, *a typical model transformation rule is executed parallelly* (in *forall* mode) for each independent matching. However, *forall* type rule applications are not directly supported by general purpose graph transformation tools. Moreover, as in most cases more than a single language is involved in transformations, the *simultaneous handling of multiple metamodels is not flexible* in these tools.
- **Verifying model transformation systems:** An ongoing research activity *integrates model transformation systems with existing model checking tools* for formal verification purposes which requires that the Kripke automata of the system is derived from the same (intermediate) semantic representation as the automatically generated target program. As a benchmark application, we generate SAL [2] specifications from UML Statecharts, where statecharts semantics are captured by model transformation systems.

Acknowledgment

The author would like to thank András Pataricza for suggesting many improvements in the early versions of the paper, and the anonymous referees for their valueable comments.

References

1. R. Bardohl and H. Ehrig. Conceptual model of the graphical editor GENGED for the visual definition of visual languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. Theory and Application to Graph Transformations (TAGT'98)*, vol. 1764 of *LNCS*, pp. 252–266. Springer, 2000.
2. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway (ed.), *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pp. 187–196. 2000.
3. A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, vol. 16(5):pp. 265–275, 2001.
4. M. Dal Cin, G. Huszerl, and K. Kosmidis. Evaluation of safety-critical systems based on guarded statecharts. In R. Paul and C. Meadows (eds.), *Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering.* IEEE, 1999.
5. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.
6. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *In [15]*, chap. Algebraic Approaches to Graph Transformation — Part II: Single pushout approach and comparison with double pushout approach, pp. 247–312. World Scientific, 1997.
7. G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In R. France and B. Rumpe (eds.), *Proc. UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference*, vol. 1723 of *LNCS*, pp. 473–488. Springer, 1999.
8. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. Theory and Application to Graph Transformations (TAGT'98)*, vol. 1764 of *LNCS*, pp. 296–309. Springer, 2000.
9. C. Kobryn. UML 2001: A standardization odyssey. *Communications of the ACM*, vol. 42(10), 1999.
10. O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In *GRATRA 2000, Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pp. 32–39. 2000.
11. D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, vol. 11(6):pp. 637–664, 1999.
12. U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE).* ACM Press, Limerick, Ireland, 2000.
13. Z. Pap, I. Majzik, and A. Pataricza. Checking general safety criteria on UML statecharts. In U. Voges (ed.), *Proc. SAFECOMP 2001, Computer Safety, Reliability and Security, 20th International Conference*, vol. 2187 of *LNCS*, pp. 46–55. Springer, 2001.
14. D. Peled. *Software Reliability Methods*, chap. Message Sequence Charts, pp. 300–302. Springer, 2001.

15. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations*, vol. 1: Foundations. World Scientific, 1997.
16. A. Schürr. *In [15]*, chap. Programmed Graph Replacement Systems, pp. 479–546. World Scientific, 1997.
17. A. Schürr, A. J. Winter, and A. Zündorf. *In [5]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.
18. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In J. Padberg (ed.), *UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, vol. 44 of *ENTCS*. 2001.
19. D. Varró and A. Pataricza. Mathematical model transformations for system verification. Tech. rep., Budapest University of Technology and Economics, 2001. `http://www.inf.mit.bme.hu/~varro/publication/publication.htm`.
20. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*. In print.

Diagonal Flip Operations on Realizers and Their Application to Wagner's Theorem

Nicolas Bonichon, Bertrand Le Saëc and Mohamed Mosbah

LaBRI-Université Bordeaux 1, 351 Cours de la Libération,
33405 Talence, France
{bonichon, lesaec, mosbah}@labri.fr

Abstract. A realizer of a maximal plane graph is a set of three particular spanning trees. It has been used in several graph algorithms and particularly in graph drawing algorithms. We propose colored flips on realizers to generalize Wagner's theorem on maximal planar graphs to realizers. From this result, it is proved that $\xi_0 + \xi_1 + \xi_2 - \Delta = n - 1$ where ξ_i is the number of inner nodes in the tree T_i, Δ is the number of three colored faces in the realizer and n is the number of vertices. As an application of this formula, we show that orderly spanning trees with at most $\lfloor \frac{2n+1-\Delta}{3} \rfloor$ leaves can be computed in linear time.

1 Introduction

Schnyder showed that every maximal plane graph admits a special decomposition of its interior edges into three trees, called realizer [13, 14]. Such a decomposition can be constructed in linear time [14]. Using realizers, it has been proved in [14] that every plane graph with $n \geq 3$ vertices has a planar straight-line drawing in a rectangular grid area $(n-2) \times (n-2)$. The existence of straight line embeddings for planar graphs was independently proven by Wagner [16], Fary [8] and Stein [15]. However, the question whether a grid of polynomial size for such an embedding exists was raised by Rosenstiehl and Tarjan [12].

Realizers are useful for many graph algorithms, of course for graph drawing [14, 2] but also for graph encoding [3]. They are strongly connected with canonical ordering (or shelling order) [9, 11], with 3-orientations [4], and with orderly spanning trees [2]. They can also be used to characterize planar graphs in terms of the order of their incidence, i.e., a graph G is planar iff the dimension of the incidence order of vertices and edges is at most 3 [13]. Realizers of the same graph have already been investigated. Suitable operations transforming a realizer of a graph to another realizer of the same graph have been investigated in [4, 1]. A particular normal form is also characterized. Moreover, the structure of the set of realizers of a given graph turns out to be a distributive lattice [4].

In this paper, we deal with realizers of size n, i.e. realizers of maximal plane graphs of size n. Wagner [16] showed that any two maximal planar graphs having the same number of vertices are equivalent under diagonal flip transformations. A diagonal flip is a graph operation on maximal planar graphs which consists of removing the diagonal (u_1, u_3) of a quadrilateral (u_1, u_2, u_3, u_4), and inserting the opposite diagonal (u_2, u_4) (see Fig. 4). Hence, one can obtain all maximal plane graphs of size n using diagonal flips. An extension of Wagner's result to torus graphs has been proved in [5]. We generalize Wagner's theorem to realizers. Moreover, we show that the set of all realizers is a poset, i.e. a partially ordered set, with some appropriate relationship. Instead of the flip operation, we introduce two new operations, that we will call colored or oriented flips which will be used to get a realizer from another one of the same size.

Flip operations are also related to the four color theorem. In [6], a signed version of the diagonal flip has been used to define transformations between signed triangulations of a polygon. The existence of a sequence of signed flips between two given triangulations of a polygon turns out to be equivalent to the four color theorem [6, 7, 10]. Our approach involves an "oriented" version of diagonal flips.

As an application of our main result, we characterize the number of inner nodes of realizers. Precisely, we prove that $\xi_0 + \xi_1 + \xi_2 - \Delta = n - 1$ where ξ_i is the number of inner nodes in the tree T_i and Δ is the number of three colored faces in the realizer. As an application of this result, we prove that an orderly spanning tree with at most $\lfloor \frac{2n+1-\Delta}{3} \rfloor$ leaves can be computed in linear time for a maximal plane graphs. This bound is a precise formulation of that given in [2].

The rest of this paper is organized as follows. In Section 2, we present realizers and we give some basic properties. An extension of Wagner's theorem on realizers and the structure of the set of realizers are investigated in Section 3. We prove in Section 4 that $\xi_0 + \xi_1 + \xi_2 - \Delta = n - 1$, and we apply it to orderly spanning trees. Section 5 concludes the paper.

2 Preliminaries

2.1 Definitions

We assume that the reader is familiar with graph theory. In this paper we deal with simple graphs. A drawing of a graph is a mapping of each vertex to a point of the plane and of each edge to the continuous curve joining the two ends of this edge. A planar drawing or *plane graph* is a drawing without crossing edges except, possibly, on a common extremity. A graph that has a planar drawing is a planar graph. A plane graph splits the plane into topologically connected regions, called *face regions*. A *face* is the counterclockwise walk of the boundary of a face region. One of the regions is unbounded and its associated face is named the *external face* of the plane graph. The vertices and edges of this face are also qualified as *external*, the other vertices are called *inner* ones. A *cycle* is an eulerian connected partial subgraph (i.e. with vertices of even degree only). A cycle is *elementary* if all its vertices are of degree 2. A *circuit* is a cycle where each vertex has an out-degree equal to its in-degree. A *k-cycle* is a cycle of k edges. Since we deal with plane graphs, a cycle defines an interior region. For simplicity, we say the *region* C, for the region delimited by the cycle C.

A planar graph G is *maximal* (or *triangulated*) if all the other graphs with the same number of vertices that contain G, are not planar. The faces of a maximal plane graph are triangular. In this case, we denote v_0, v_1, v_2 the three vertices of the external face of this plane graph.

Definition 1 *[13] A realizer of a maximal plane graph G is a partition of the interior edges of G in three sets T_0, T_1, T_2 of directed edges such that for each interior vertex u it holds thats*

1. *u has out-degree exactly one in each of T_0, T_1, T_2.*
2. *The counterclockwise order of the edges incident with u is: leaving in T_0, entering in T_2, leaving in T_1, entering in T_0, leaving in T_2 and entering in T_1 (see Fig. 1).*

So a realizer is a set of three rooted trees where their edges are oriented to their roots, which are the external vertices v_0, v_1, v_2. For simplicity, we write $i+1$ as

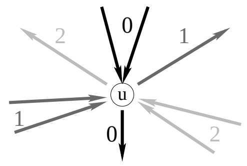


Fig. 1. Edge coloration and orientation around a vertex

a shorthand for $(i+1) \mod 3$ and $i-1$ as a shorthand for $(i+2) \mod 3$. In the rest of the article, we assume that the edges of the tree T_i are colored with color i, where $i \in \{0, 1, 2\}$ and that the external edges (v_i, v_{i+1}) are of the color $i+1$. In fact, we consider a slightly different definition of realizers by coloring the external edges in order to reduce the number of particular cases.

An example of a graph, and a realizer of this graph are given in Fig. 2.

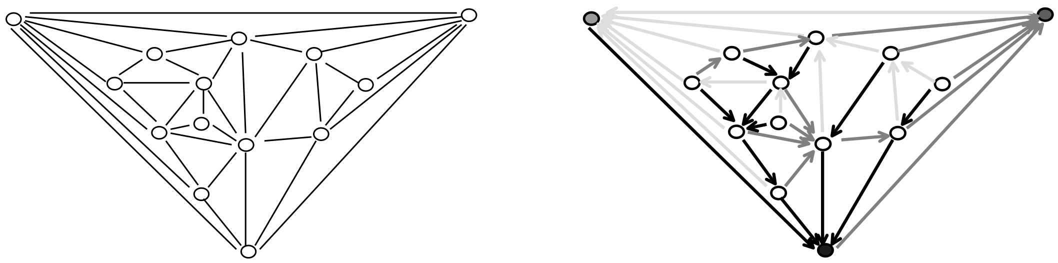


Fig. 2. An example of a realizer (a graph on the left side, and one of its realizer on the right side)

We denote by $deg_i(v)$ the number of ingoing edges of v in T_i. $u_1 \xrightarrow{i} u_2$ denotes the path colored i from u_1 to u_2. We write $u_1 >^i_{ccw} u_2$ (resp. $u_1 >^i_{cw} u_2$) if u_1 is after u_2 in the *counterclockwise preordering* (resp. *clockwise preordering*) of the tree T_i. Counterclockwise (resp. clockwise) preordering of a tree means visiting the root, then recursively the subtrees in the counterclockwise (resp. clockwise) order.

2.2 Properties of Realizers

Let $F = (e_0, e_1, e_2)$ be a face of G with $e_j = \{u_j, u_{j+1}\}$.

Property 1 *If u_1 is the parent of u_2 in T_i then $u_1 >^{i+1}_{cw} u_2$ and $u_2 >^{i-1}_{cw} u_1$.*

The proof of this property is based on the following facts.

Fact 1 *Let e_0 be colored i.*
If e_0 and e_1 are oriented towards u_1 then e_1 is colored i.

Proof. If e_1 is not colored i, the parent of u_1 in T_{i+1} would be inside the face F. This is not possible.

Fact 2 *Let e_0 be colored i.*
If e_0 and e_1 are respectively oriented towards u_1 and u_2 then e_1 is colored $i+1$. Similarly, if e_0 and e_2 are respectively oriented towards u_0 and u_1 then e_2 is colored $i-1$.

Proof. Assume that e_0 and e_1 are respectively oriented towards u_1 and u_2. If e_1 is not colored $i+1$, the parent of u_1 in T_{i+1} would be inside the face F. This is not possible. A similar argument can be used to prove the second part of the fact.

As a consequence of the above facts, there are four possible colorations of a face which are given in Fig. 3. Notice that the last two colorations use the three colors. We will say the faces are *three-colored* ones.

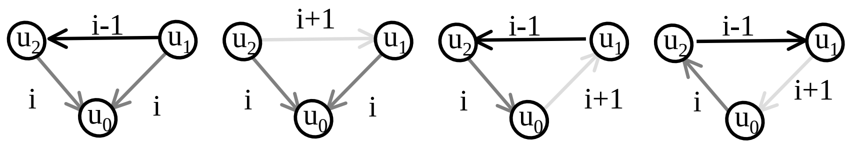


Fig. 3. Edge coloration of a face

2.3 Realizer distributive lattice

In this section, we recall results on the set of the realizers of a given graph. In particular, its structure is a distributive lattice.

Since we work with embedded graphs, a directed cycle is *oriented* either clockwise or counterclockwise, also noted as *cw-cycle* (resp. *ccw-cycle*).

Lemma 1 *[1] Let R be a realizer of a maximal plane graph G. Let C be a ccw-cycle (resp. cw-cycle) in R. Then we obtain a new realizer R' of G by*

1. *reversing the direction of the edges of C*
2. *setting for each edge of C the new color to be the color succeeding (resp. preceding) its original color,*
3. *setting for each edge inside C the new color to be the color preceding (resp. succeeding) its original color,*
4. *leaving the color of any other edge unchanged.*

Definition 2 *[1]*

- *Let G be a maximal plane graph. Define $\mathcal{R}(G) := \{R\text{: } R \text{ is a realizer of } G\}$.*
- *For two realizers R_1 and R_2 of G, $R_1 \preceq R_2$ iff R_1 can be obtained from R_2 by re-coloring some cw-cycles, i.e. there exists a re-coloration sequence which transforms R_1 into R_2.*
- *Let L_G (resp. R_G) be the realizer of G without any cw-cycle, (resp. ccw-cycle).*

Theorem 1 *[1] Let G be a maximal plane graph. $(\mathcal{R}(G), \preceq)$ is a distributive lattice.*

In the following section, we will deal with the set of all realizers of size n.

3 Diagonal flips on realizers

3.1 Diagonal flip

In [16], Wagner proved that it is possible to obtain all maximal planar graphs of size n using a graph rewriting rule called diagonal flip. In this section, we extend this result to realizers using colored flips.

Definition 3 *Let G be an embedded graph. Let u_2, u_1, u_4 and u_3, u_4, u_1 be two adjacent faces where u_2 is not neighbor of u_3. A diagonal flip consists of removing the edge (u_1, u_4) and inserting the edge (u_2, u_3) (see Fig. 4).*

Theorem 2 *[16] Let G_1 and G_2 be two maximal planar graphs with n vertices. There exists a sequence of diagonal flips which transforms G_1 into G_2.*

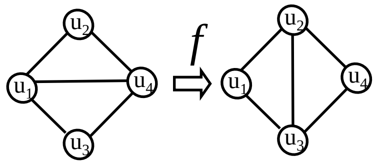


Fig. 4. Diagonal flip operation

3.2 Generalization to realizers

As shown in Fig. 5, we propose colored diagonal flips for realizers using two kinds of flips: f_1^i and f_2^i. It is easy to see that the application of a diagonal flip f_1^i or f_2^i on a realizer gives another realizer.

The choice between f_1^i and f_2^i depends of the quadrilateral configuration. Note that if the edge (u_2, u_1) is colored $i-1$ and oriented towards u_1, and if the edge (u_3, u_1) is colored $i+1$ and oriented towards u_1, then f_1^i or f_2^i can be applied.

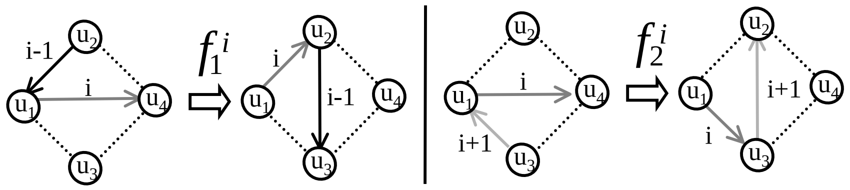


Fig. 5. Flips on realizer

Unfortunately, it is not always possible to apply one of the two operations. This occurs for the configuration of the quadrilateral of Fig. 6.

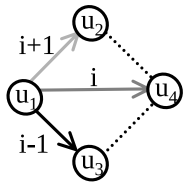


Fig. 6. Configuration for which colored flip cannot be directly applied

Now, we define two orders on trees, $\leq_{cw}$ and $\leq_{ccw}$, that are useful to express some flip properties.

Definition 4 *Let T and T' be ordered rooted trees with k nodes. Let $n_1, n_2, ..., n_k$ and $m_1, m_2, ..., m_k$ be the nodes of T and T' in the clockwise preordering (resp. counterclockwise preordering). If $T = T'$ then $T \leq_{cw} T'$ and $T \leq_{ccw} T'$. Else,* *let i be the rank of the first node where $deg(n_i) \neq deg(m_i)$. If $deg(n_i) < deg(m_i)$ then $T \leq_{cw} T'$ (resp. $T \leq_{ccw} T'$).*

Naturally, $T \geq_{cw} T'$ means that $T' \leq_{cw} T$. Also, $T <_{cw} T'$ means that $T \leq'_{cw} T$ and $T \neq T'$. If we consider the example of Fig. 7, we can see that the two grey nodes are the first nodes in T and T' with different degrees, with respect to the clockwise preordering. As the grey node in T has more children than in T', $T >_{cw} T'$. Similarly, the two black nodes are the first which have different degrees in T and T', in the counterclockwise preordering. Since the black node in T' has more children than that in T, $T <_{ccw} T'$.

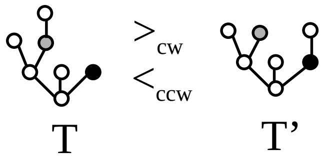


Fig. 7. Illustration of order relation between trees

Property 2 *Let $R = (T_0, T_1, T_2)$ be a realizer. Let $R' = (T'_0, T'_1, T'_2)$ be a realizer obtained from R by applying a flip f_1^i (resp. f_2^i). We have the following properties: $T'_i <_{cw} T_i$, $T'_{i-1} >_{ccw} T_{i-1}$ (resp. $T'_i <_{ccw} T_i$, $T'_{i+1} >_{cw} T_{i+1}$)*

Proof. Let us consider the flip f_1^i of Fig. 5. The edge (u_1, u_3) can be colored $i-1$ and oriented towards u_3 or colored $i+1$ and oriented towards u_1. In both cases $u_1 <_{ccw}^{i-1} u_3$. As the number of children of u_1 is greater in T'_{i-1} than in T_{i-1}, $T'_{i-1} >_{ccw} T_{i-1}$. The edge (u_2, u_4) can be colored i and oriented towards u_4 or colored $i+1$ and oriented towards u_2. In both cases, $u_4 <_{cw}^{i} u_2$. Since the number of children of u_4 is lesser in T'_i than in T_i, $T'_i <_{cw} T_i$.

3.3 Structure of $\mathcal{R}_n$ and Wagner's theorem

Let $\mathcal{R}_n$ be the set of realizers of graphs of size n. The set of all realizers of size n can be represented by an oriented colored graph where vertices stand for realizers, and an edge colored i between two vertices R and R' represents the flip f_1^i transforming R into R'. Fig. 8 shows the graph of realizers of size 6. Each vertex represents a realizer. There is a directed edge colored i from a realizer R to another one R' if R' can be obtained from R by a flip f_1^i. The right part of the figure displays the transformation of the realizer 6 into the realizer 5 by a flip f_1^0.

We write $R(f_1^i|f_1^{i+1})^*R'$ if R can be transformed into R' by a sequence of flips f_1^i and f_1^{i+1}. Let $(\mathcal{R}_n, f_1^i|f_1^{i+1})$ be the set of realizers of size n, equipped with the relation $(f_1^i|f_1^{i+1})^*$.

Property 3 $(\mathcal{R}_n, f_1^i|f_1^{i+1})$ *is a poset, i.e. a partially ordered set.*

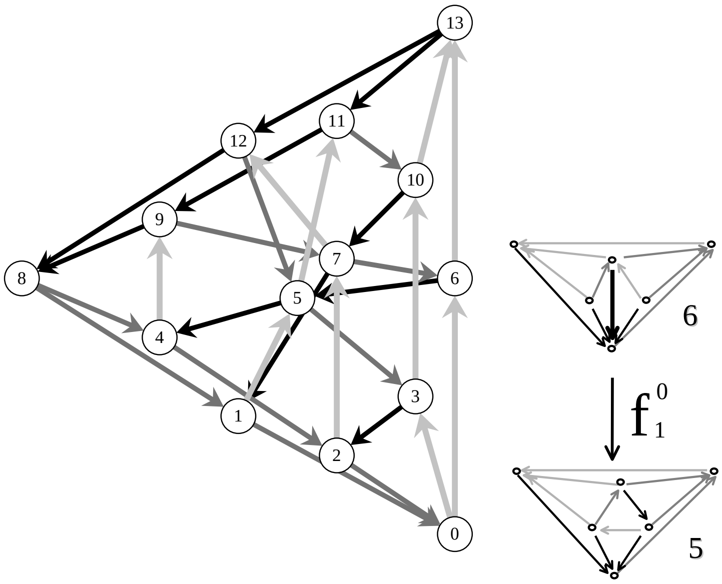


Fig. 8. $\mathcal{R}_6$ with f_1 operations.

Proof. When applying f_1^i, $T_i' <_{cw} T_i$ and $T_{i-1}' >_{ccw} T_{i-1}$. When applying f_1^{i+1}, $T_{i+1}' <_{cw} T_{i+1}$ and $T_i' >_{ccw} T_i$. So, using the flips f_1^i or f_1^{i+1}, T_{i-1} is strictly decreasing considering the order $>_{ccw}$ and T_{i+1} is strictly increasing considering the order $>_{cw}$. Hence $(f_1^i|f_1^{i+1})^*$ is antisymmetric. An empty sequence transforms R into R, and thus the relation is reflexive. The relation is also transitive since flip sequences can be concatenated.

Let D_n^i be the realizer of size n where all the inner vertices are on the same branch of T_i and T_{i+1} and T_{i-1} are trees of depth 1 (see Fig. 9).

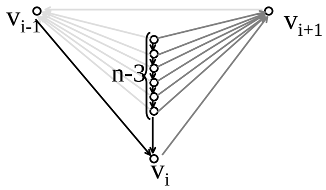


Fig. 9. Realizer D_n^i

Property 4 *D_n^{i-1} is the upper bound of $(\mathcal{R}_n, f_1^i|f_1^{i+1})$ and D_n^{i+1} is the lower bound. D_n^{i-1} is the lower bound of $(\mathcal{R}_n, f_2^{i-1}|f_2^i)$ and D_n^{i+1} is the upper bound.*

Proof. Let $R = (T_0, T_1, T_2)$ be a realizer of size n. If $R \neq D_n^{i+1}$ then there is an inner node in T_i or in T_{i-1}. If u is an inner node of T_i, a flip f_1^{i+1} can be applied on its outgoing edge colored $i+1$. Similarly, if u is an inner node of T_{i-1}, a flip f_1^i can be applied on its outgoing edge colored i. Hence a flip f_1^i or a flip f_1^{i+1} can be applied. In the same way, we show that D_n^{i-1} is the upper bound of $(\mathcal{R}_n, f_1^i|f_1^{i+1})$. The second part of the property comes directly from the fact that f_2^{i-1} is the inverse of f_1^i and f_2^i is the inverse of f_1^{i+1}.

Theorem 3 *There exists a sequence of colored flips that transforms any realizer R with n vertices into any other realizer R' with n vertices.*

Proof. Let R and R' be two realizers with n vertices. As D_n^{i-1} is the unique maximal element of $(\mathcal{R}_n, f_1^i|f_1^{i+1})$, there is a flip sequence that transforms R into D_n^{i-1}. There is also a flip sequence that transforms D_n^{i-1} into R'. Hence, the concatenation of the two previous sequences transforms R into R'.

In the previous theorem, flips f_1 and f_2 are allowed to transform a realizer to another one. It is possible to use only flips f_1 as stated in the following corollary.

Corollary 1 *There exists a sequence of colored flips $(f_1^0|f_1^1|f_1^2)^*$ that transforms any realizer R with n vertices into any other realizer R' with n vertices.*

Proof. Since D_n^0 is the upper bound of $(\mathcal{R}_n, f_1^1|f_1^2)$ there exists a flip sequence $(f_1^1|f_1^2)^*$ that transforms R into D_n^0. Since D_n^0 is the upper bound of $(\mathcal{R}_n, f_2^1|f_2^2)$, there exists a sequence $(f_2^1|f_2^2)^*$ that transforms R' into D_n^0. The inverse flip sequence transforms D_n^0 into R' and is composed of flips f_1^0 and flips f_1^1. Hence, the concatenation of the two appropriated sequences gives a flip sequence $(f_1^0|f_1^1|f_1^2)^*$ that transforms R into R'.

4 Three-colored faces and number of inner nodes in realizers

Let Δ be the number of three-colored faces of a realizer R. Let ξ_i be the number of inner nodes of the tree T_i.

Lemma 2 *Let R be a realizer. Let R' be a realizer obtained from R with a flip f_1^i. The sum $\xi_0 + \xi_1 + \xi_2 - \Delta$ is the same for R and R'.*

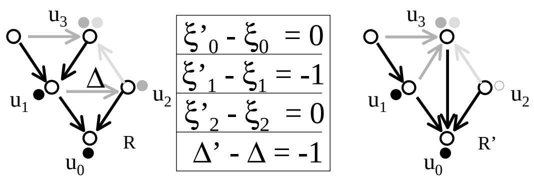


Fig. 10. Example of flip configuration

Proof. To prove the lemma, we need to check the modifications induced by the flip f_1^i on the quadrilateral and the adjacent face of the edge (u_1, u_3). If we consider the configuration of Fig. 10, we can see that u_2 is an inner node of T_1 but not of T_1'. Moreover, R has a three-colored face whereas R' does not. So the lemma holds for this configuration.

More generally, there are 32 possible configurations of the quadrilateral and the adjacent face of the edge (u_1, u_3) for the flip f_1^i. Fig. 11 shows the 32 possible configurations. For all these configurations, the lemma is verified.

Theorem 4 *For any realizer R of a maximal plane graph G, we have $\xi_0 + \xi_1 + \xi_2 - \Delta = n - 1$.*

Proof. Since all realizers of size n can be obtained from one another by applying operations f_1 which preserve the sum $\xi_0 + \xi_1 + \xi_2 - \Delta$, all realizers of size n have the same sum. In the particular case of the realizer D_n^0 of Fig. 9, this sum is equal to $n - 1$ ($\xi_0 = n - 3, \xi_1 = 1, \xi_2 = 1$ and $\Delta = 0$).

The *orderly spanning tree* [2] of a maximal plane graph can be obtained from a Schnyder tree T_i by adding the edge (v_{i+1}, v_i) to T_i.

Corollary 2 *Let R be a realizer of a plane graph G. An orderly spanning tree can be obtained in linear time from R with at most $\lfloor \frac{2n+1-\Delta}{3} \rfloor$ leaves.*

Proof. This corollary follows from the fact that the number of leaves in a Schnyder tree T_i is $n - 1 - \xi_i$ and that the orderly spanning tree obtained from T_i has one more leaf, the node v_{i+1}.

5 Conclusion and remarks

In this paper, we proved Wagner's theorem for realizers of maximal plane graphs of size n. The bound for the number of leaves of an orderly spanning tree, given in Corollary 2, improves that given in [2]. This result can be useful particularly for classes of graphs where $\Delta > 0$, such as 4-connected graphs. As an application of this bound, there exists an $n \times (\frac{2n+1-\Delta}{3})$ grid for drawing a plane graph of size n.

Besides, we have shown that $(\mathcal{R}_n, f_1^i | f_1^{i+1} | f_2^{i+1})$ has a structure of a bounded poset. We have also looked for richer structure such as lattices. Unfortunately, neither $(\mathcal{R}_6, f_1^i | f_1^{i+1} | f_2^{i+1})$, $(\mathcal{R}_6, f_1^i | f_1^{i+1})$ nor $(\mathcal{R}_6, f_1^i | f_2^{i+1})$ has a structure of a lattice. We conjecture that $(\mathcal{R}_n, f_1^i | f_1^{i+1} | f_2^{i+1})$ is a poset. We have shown that this conjecture is true for $n \leq 6$.

Acknowledgments

The authors would like to thank Sylvain Gravier for his helpful remarks.

References

1. E. Brehm. *3-orientations and Schnyder 3-tree-decompositions.* PhD thesis, FB Mathematik und Informatik, Freie Universität Berlin, 2000.
2. Yi-Ting Chiang Ching-Chi and Hsueh-I Lu. Orderly spanning trees with applications to graph encoding and graph drawing. In *Proc. 12th Symp. Discrete Algorithms*, pages 506–515. ACM and SIAM, 2001.
3. Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical ordering and multiple parentheses. In *Proc. 25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, volume 1443, pages 118–129, 1998.
4. P. Ossona de Mendez. *Orientations bipolaires.* PhD thesis, Ecole des Hautes Etudes en Sciences Sociales, 1994.
5. A. K. Dewdney. Wagner's theorem for torus graphs. *Discrete Math.*, 4:139–149, 1973.
6. S. Eliahou. Signed diagonal flips and the four color theorem. *Europ. J. Combinatorics*, 20:641–646, 1999.
7. S. Eliahou, S. Gravier, and C. Payan. Three moves on signed surface triangulations. *Les cahiers du laboratoire leibniz*, 2000.
8. I. Fary. On straight lines representation of planar graphs. *Acta Sci. Math. Szeged*, 11:229–233, 1948.
9. H. De Frayseix, J. Pach, and J. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.
10. S. Gravier and C. Payan. Flips signés et triangulations d'un polygone. *Les cahiers du laboratoire leibniz*, 2000.
11. G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16:4–32, 1996.
12. P. Rosenstiehl and R.E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete Comput. Geom.*, 1:343–353, 1986.
13. W. Schnyder. Planar graphs and poset dimension. *Order*, 5:323–343, 1989.
14. W. Schnyder. Embedding planar graphs on the grid. *Proc. 1st ACM-SIAM Symp. Discrete Algorithms*, pages 138–148, 1990.
15. S.K. Stein. Convex maps. In *Proc. Amer. Math. Soc.*, volume 2, pages 464–466, 1951.
16. K. Wagner. Bemerkungen zum Vierfarbenproblem. In *Jahresber. Deutsche Math. -Verein.*, volume 46, pages 26–32, 1936.

Appendix

A colored version of the configurations is available at the following URL:
`http://dept-info.labri.fr/~bonichon/Realizer`.

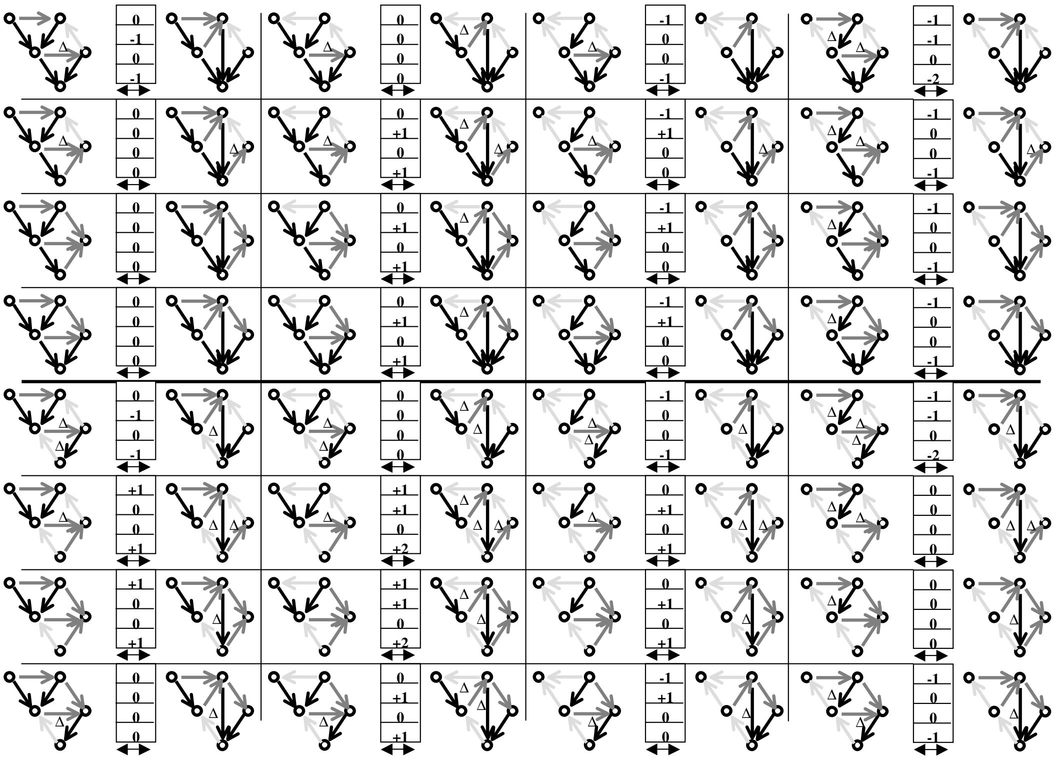


Fig. 11. 32 configurations of flips f_1

A Core Language for Graph Transformation

Annegret Habel[1] and Detlef Plump[2]

[1] Fachbereich Informatik, Universität Oldenburg
Postfach 2503, D-26111 Oldenburg, Germany
habel@informatik.uni-oldenburg.de

[2] Department of Computer Science, The University of York
York YO10 5DD, United Kingdom
det@cs.york.ac.uk

Abstract. We show how to program in a core programming language based on graph transformation rules. Our programs compute various functions and relations on graphs: the functions generating the transitive closure of a graph and the disjoint union of all subgraphs of a graph, the relation yielding all spanning trees of a graph, and functions testing for connectedness, acyclicity, and planarity. The language has a simple syntax consisting of just three constructs: nondeterministic one-step application of a set of rules, sequential composition, and iteration. It also has a simple formal semantics, and was shown to be computationally complete and minimal.

1 Introduction

In [3] we introduced a programming language for graph transformation consisting of three constructs: nondeterministic application of a rule from a set of graph transformation rules (according to the double-pushout approach), sequential composition of programs, and iteration in the form that a program is applied as long as possible. The language has a simple formal semantics and is computationally complete in that it allows to compute every computable partial function on labelled graphs. Moreover, the language is minimal in that omitting either sequential composition or iteration results in an incomplete language.

In this paper we show how to program in this language, by giving programs for the following problems: generating the transitive closure of a graph, generating the disjoint union of all subgraphs of a graph, generating all spanning trees of a graph, and testing whether a graph is connected, acyclic, or planar, respectively. The test functions are non-destructive in that they preserve the input graph, hence their programs can be used as components of programs which do further computations.

We consider the proposed programming language as a (declarative) core language because it lacks types and high-level constructs like procedures and modules. Having a simple yet computationally complete core language has several advantages:

– High-level languages for graph transformation which provide more programming comfort can be defined by mapping high-level constructs to the core. In this way possibly complex languages can be obtained which by their definition automatically get a formal semantics. Moreover, new languages are known to be computationally complete if they just cover the core language.
– Implementations for new languages based on the core can be rapidly obtained by extending an implementation of the core with translations from high-level constructs into core constructs.
– Frameworks and systems for formal reasoning on graph programs can be restricted to deal with the core language, since high-level programs can be translated into semantically equivalent programs in the core language. For example, frameworks for both program verification and program transformation can benefit from this approach.

The next section reviews the syntax and semantics of the core language and mentions computational completeness and minimality. Section 3 constitutes the main part of this paper, showing how to program in the language by means of several examples. Finally, two auxiliary program schemes implementing a copy operation and a conditional statement are given in the Appendix.

2 The language

Programs are based on sets of graph transformation rules according to the double-pushout approach, where rules are matched injectively and may have non-injective right-hand morphisms. See [3] and [2] for details.

Definition 5 (Syntax). *Programs* over a label alphabet $\mathcal{C}$ are inductively defined as follows:

(1) Every finite set $\mathcal{R}$ of rules over $\mathcal{C}$ is a program.
(2) If P_1 and P_2 are programs, then $\langle P_1; P_2\rangle$ is a program.
(3) If P is a program according to (1) or (2), then $P\!\downarrow$ is a program.

Programs according to (1) are *elementary*, the program $\langle P_1; P_2\rangle$ is the *sequential composition* of P_1 and P_2, and $P\downarrow$ is the *iteration* of P. Programs of the form $\langle P_1; \langle P_2; P_3\rangle\rangle$ and $\langle\langle P_1; P_2\rangle; P_3\rangle$ are considered as equal and can both be written as $\langle P_1; P_2; P_3\rangle$; this is justified in that sequential composition is semantically the composition of binary relations, which is associative (see below).

We consider graph transformation over abstract graphs (isomorphism classes of graphs), denoting by $\mathcal{A}_\mathcal{C}$ the set of all abstract graphs over a label alphabet $\mathcal{C}$. Given a binary relation $\to$ on a set S, we denote by $\to^+$ the transitive closure of $\to$ and by $\to^*$ the reflexive-transitive closure. The *domain* of $\to$, denoted by $\mathrm{Dom}(\to)$, consists of all elements a in S such that $a \to b$ for some b.

Definition 6 (Semantics). Given a program P over a label alphabet $\mathcal{C}$, the *semantics* of P is a binary relation $\to_P$ on $\mathcal{A}_\mathcal{C}$ which is inductively defined as follows:

(1) $\to_P \,=\, \Rightarrow_\mathcal{R}$ if P is an elementary program $\mathcal{R}$.
(2) $\to_{\langle P_1;P_2\rangle} \,=\, \to_{P_1} \circ \to_{P_2}$.
(3) $\to_{P\downarrow} \,=\, \{\langle G,H\rangle \mid G \to_P^* H \text{ and } H \notin \mathrm{Dom}(\to_P)\}$.

Consider now subalphabets $\mathcal{C}_1$ and $\mathcal{C}_2$ of $\mathcal{C}$ and a relation $Rel \subseteq \mathcal{A}_{\mathcal{C}_1} \times \mathcal{A}_{C_2}$. We say that P *computes Rel* if $Rel = \to_P \cap (\mathcal{A}_{C_1} \times \mathcal{A}_{\mathcal{C}_2})$, that is, if Rel coincides with the semantics of P restricted to $\mathcal{A}_{\mathcal{C}_1}$ and $\mathcal{A}_{C_2}$. The same applies to partial functions $f\colon \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{C_2}$, which are just special relations.

We remark that our programs can be formulated as semantically equivalent *graph transformation units* in the sense of [6]. Hence the results stated below apply to a certain sublanguage of graph transformation units, too.

Next we mention results from [3] on the computational completeness and the minimality of our language, without defining when a partial function on graphs is computable. The definition is based on an encoding of graphs as expressions. Intuitively, a partial function f on abstract graphs is computable if there is a computable function f' on strings such that for every abstract graph G for which f is defined and every graph expression w denoting G, f' is defined for w and yields a graph expression denoting $f(G)$. Moreover, f' is not defined on graph expressions denoting graphs on which f is not defined.

Theorem 1 (Completeness). *For every computable partial function f on abstract graphs there exists a program that computes f.*

The language is also minimal, meaning that omitting either sequential composition or iteration results in a computationally incomplete language.

Theorem 2 (Minimality).

1. *The set of programs without sequential composition is computationally incomplete.*
2. *The set of programs without iteration is computationally incomplete.*

For example, in [3] it is shown that the function converse: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_C$ which swaps source and target of each edge in a graph, is not computable by any program in the above two program classes.

3 Programming

In this section we show how to use our language for solving the following graph problems: generating the transitive closure of a graph, generating the disjoint union of all subgraphs of a graph, generating all spanning trees of a graph, and testing for connectedness, acyclicity, and planarity.

The programs below make extensive use of rules that relabel nodes, although the double-pushout approach is usually formulated for totally labelled graphs and label-preserving graph morphisms which prevent such rules. We employ a generalized form of rules where nodes in the interface can be unlabelled and morphisms can send unlabelled nodes to labelled nodes [4]. It can be shown, however, that for every rule r of generalized type there is a program $P(r)$ using ordinary rules such that $\rightarrow_{\{r\}} = \rightarrow_{P(r)}$.

We display rules by showing their left- and right-hand sides, using the convention that the interface graph consists of all numbered nodes.

3.1 Generating the transitive closure

The transitive closure of a graph G is obtained by adding an edge from a node u to another node v, whenever there is in G a directed path from u to v but no direct link. The function trans: $\mathcal{A}_\mathcal{C} \rightarrow \mathcal{A}_\mathcal{C}$, assigning to every graph its transitive closure, is computed by the following program:[1]

$$\texttt{TransClosure} = \underline{\text{if}}\ \emptyset\ \underline{\text{then}}\ \texttt{TransClosure}_1\ \underline{\text{else}}\ \texttt{TransClosure}_2.$$

The subprograms $\texttt{TransClosure}_1$ and $\texttt{TransClosure}_2$ are dealing with an empty and a non-empty input graph, respectively. While $\texttt{TransClosure}_1$ does not alter the input graph, $\texttt{TransClosure}_2$ is given as follows:

$$\langle\langle\texttt{Select}_1;\ \langle\texttt{Select}_2;\ \texttt{Connect}\downarrow;\ \texttt{Forget}_3\rangle\downarrow;\ \texttt{Forget}_2\rangle\downarrow;\ \texttt{Forget}_1\rangle,$$

where $\texttt{Connect} = \langle\texttt{Select}_3;\ \texttt{Unmark}\downarrow;\ \texttt{Link}\downarrow\rangle$. The main rules of $\texttt{TransClosure}_2$ are given in Figure 1. $\texttt{Select}_1$ selects a node and gives it the index 1, $\texttt{Select}_2$

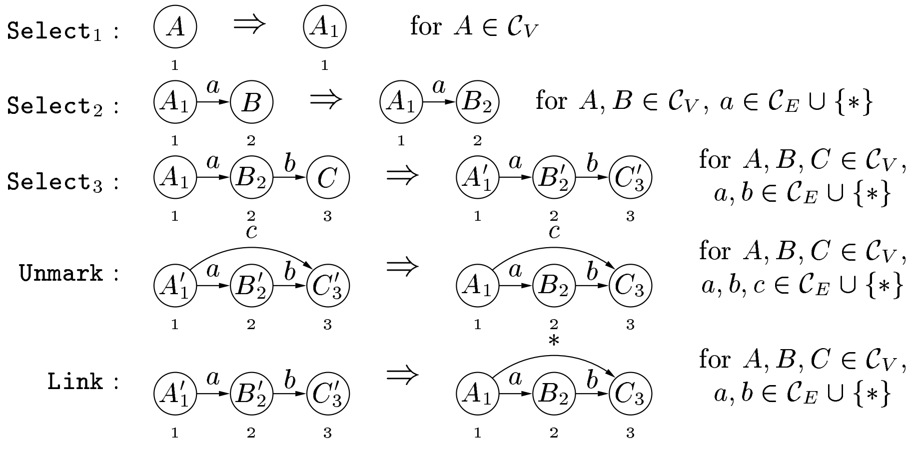


Fig. 1. The main rules of $\texttt{TransClosure}_2$

gives a neighbour of this node the index 2, and `Connect` gives a neighbour of the

[1] The program scheme " if _ then _ else _" is defined in Appendix A.1.

latter node the index 3 and, if there is no edge from the first to the third node, links these nodes. To test whether the first and the third node are already linked, `Select`$_3$ first marks the three nodes. If a link exists, then `Unmark` removes the marks; in this case `Link` is not applicable. If the first and the third node are not yet linked, then `Link` inserts an edge and removes the marks. Note that the down-arrows attached to `Unmark` and `Link` ensure that `Connect` is defined in cases where `Unmark` or `Link` is not applicable. Finally, for $i = 1, 2, 3$, the subprogram `Forget`$_i$ removes each occurrence of mark i.

3.2 Generating the disjoint union of all subgraphs

Our next program transforms an input graph into the disjoint union of all its subgraphs. `PowerGraph` will be used as a subprogram in the planarity test of subsection 3.6, but is interesting in its own right.

We consider here subgraphs that are obtained by edge deletions because only these matter for the planarity test. Extending the program to cover arbitrary subgraphs is straightforward.

In `PowerGraph` we use a subprogram `Tag` which adds a "tag" to a given graph by creating a unique auxiliary node and pointers from this node to all nodes in the graph. This form of tagging allows to identify different subgraphs of a graph. `Tag` is defined by

$$\texttt{Tag} = \langle \texttt{CreateTag};\ \texttt{LinkNode}{\downarrow};\ \texttt{Restore}{\downarrow} \rangle,$$

where the rule sets are shown in Figure 2. In these rules, the "invisible" edge label, the node label τ, and the labels $\{A^* \mid A \in \mathcal{C}_V\}$ are all fresh.

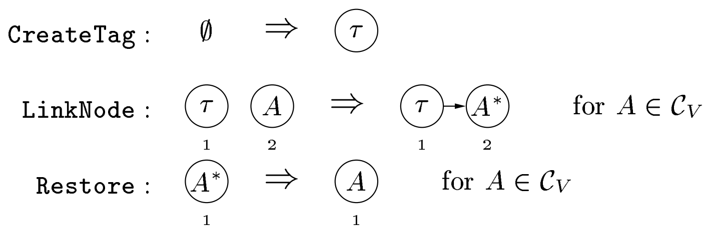


Fig. 2. The rules of `Tag`

Now `PowerGraph` is the program

$$\langle \texttt{Tag};\ \langle \texttt{PickGraph};\ \langle \texttt{PickEdge};\ \texttt{Copy};\ \texttt{RemoveEdge} \rangle{\downarrow};\ \texttt{Untag} \rangle{\downarrow};\ \texttt{CleanUp}{\downarrow} \rangle,$$

where `Copy` is an auxiliary program for copying a graph which is defined in Appendix A.2. The other subprograms are shown in Figure 3. The program `PickGraph` picks a tagged graph and removes all marks that have possibly been

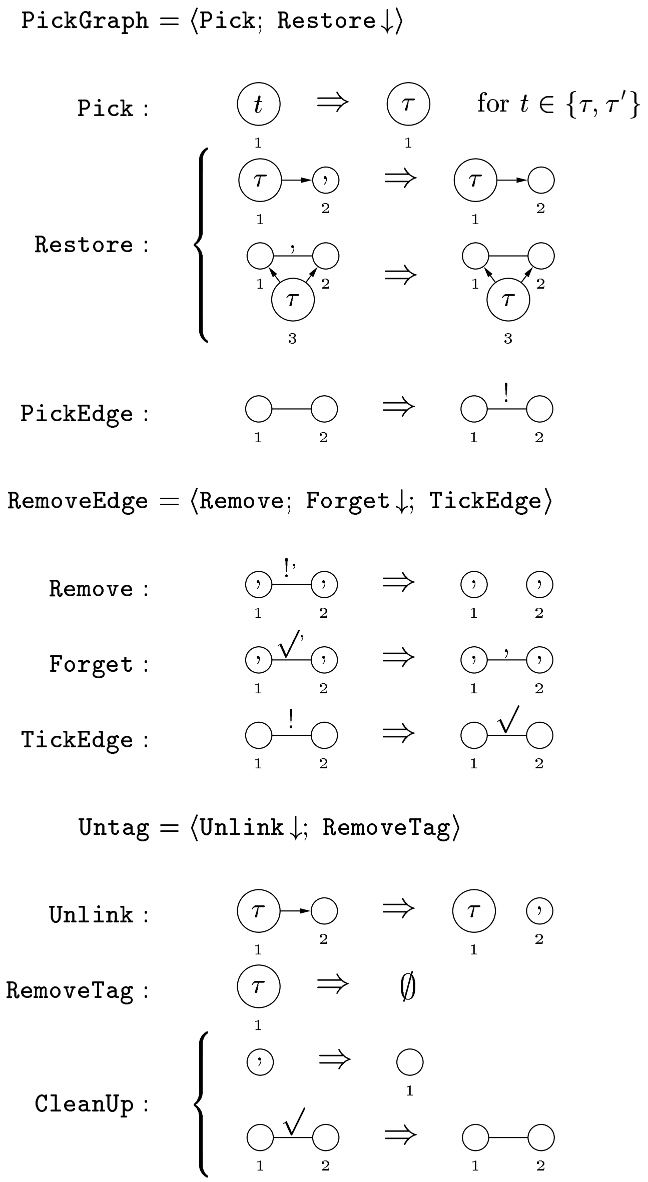


Fig. 3. Some subprograms of `PowerGraph`

created by a previous copy operation, yielding the graph "under consideration". The program `PickEdge` picks an edge in the graph under consideration and marks it with !. Then `Copy` (see Appendix A.2) copies the graph under consideration, and `RemoveEdge` removes the copy of the picked edge (marked with $!'$) and marks the picked edge with $\surd$. By repeating this process as long as possible, one obtains copies of all graphs that result from the original graph by removing a single edge. Afterwards the graph under consideration is "frozen" by removing its tag and marking all nodes with $'$. The latter prevents further copy operations on the graph. Then `PickGraph` picks a new tagged graph and the whole process starts again.

3.3 Generating all spanning trees

A subgraph S of a graph G is a spanning tree of G if the undirected graph underlying S is a tree that contains all nodes of G. Our program will "highlight" the edges of a spanning tree of a connected component of the input graph by marking them with $*$. Let Spanning $\subseteq \mathcal{A}_C \times \mathcal{A}_{C'}$ be the relation with $(G, G') \in$ Spanning if and only if G' is obtained from G by marking the edges of a spanning tree of any connected component of G with $*$. The relation Spanning is computed by the program

$$\texttt{Spanning} = \ \underline{\text{if}}\ \emptyset\ \underline{\text{then}}\ \texttt{Spanning}_1\ \underline{\text{else}}\ \texttt{Spanning}_2,$$

where $\texttt{Spanning}_1$ does not alter its input and $\texttt{Spanning}_2$ is given by

$$\texttt{Spanning}_2 = \langle \texttt{Select};\ \langle \texttt{Activate}{\downarrow};\ \texttt{Backtrack}\rangle{\downarrow};\ \texttt{Forget}\rangle.$$

This program does a depth-first search for finding a spanning tree. The search proceeds in a strictly sequential way since at any time at most one node is active, that is, marked with $\bullet$. The subprogram `Select` initially activates a node, while `Activate` activates a neighbour of the currently active node and deactivates the latter. `Backtrack` reactivates a neighbour of an active node. Finally, `Forget` forgets auxiliary node markings. The spanning tree of a component of a graph is then given by all $*$-marked edges and their incident nodes. The main rules of $\texttt{Spanning}_2$ are shown in Figure 4.

To compute spanning trees without backtracking, we can replace $\texttt{Spanning}_2$ with

$$\texttt{Spanning}'_2 = \langle \texttt{Select};\ \texttt{Activate}'{\downarrow};\ \texttt{Forget}\rangle.$$

The subprogram `Select` activates a node while $\texttt{Activate}'$ looks for an activated node, activates a neighbour, and remains active. So several nodes can be active simultaneously. As before, the spanning tree is determined by the $*$-marked edges. The main rules of $\texttt{Spanning}'_2$ are shown in Figure 5.

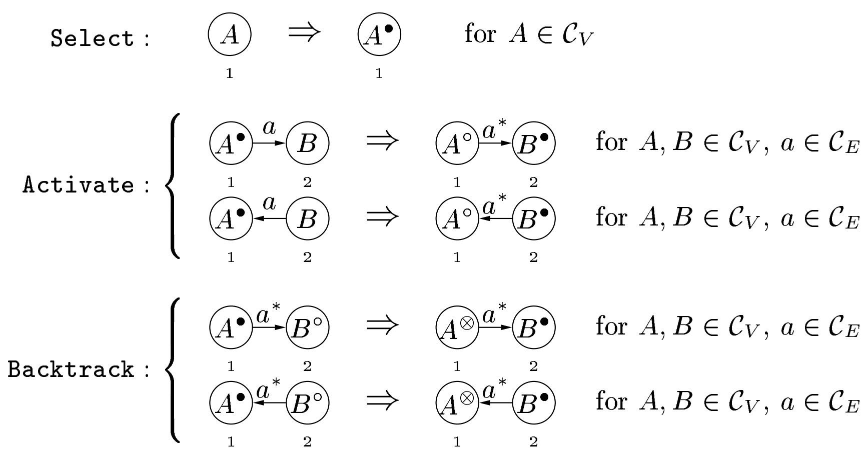


Fig. 4. The main rules of $\texttt{Spanning}_2$

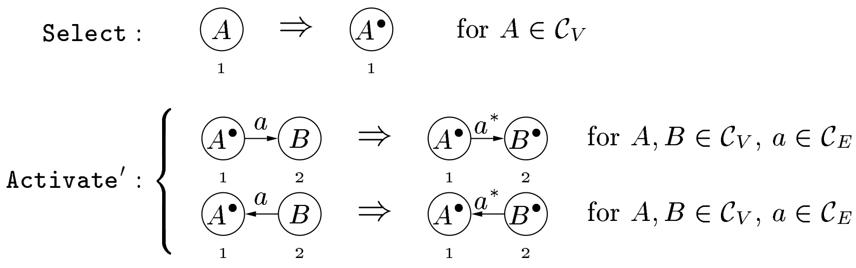


Fig. 5. The main rules of $\texttt{Spanning}'_2$

3.4 Testing for connectedness

A directed graph is connected if there is a path between each two nodes in the underlying undirected graph. The function connected?: $\mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}'}$ with

$$\text{connected?}(G) = \begin{cases} G + \text{①} & \text{if } G \text{ is connected,} \\ G + \text{⓪} & \text{otherwise} \end{cases}$$

is computed by the program

$$\texttt{Connected?} = \underline{\text{if}}\ \emptyset\ \underline{\text{then}}\ \texttt{Connected?}_1\ \underline{\text{else}}\ \texttt{Connected?}_2.$$

Here $\texttt{Connected?}_1$ creates a single node with label 1 and $\texttt{Connected?}_2$ is given by

$$\texttt{Connected?}_2 = \langle \texttt{Select};\ \texttt{Mark}\!\downarrow;\ \texttt{Check};\ \texttt{Forget}\!\downarrow \rangle.$$

The program `Select` picks any node, $\texttt{Mark}\!\downarrow$ marks all nodes that are reachable from the picked node, $\texttt{Check} = \langle \texttt{Initiate};\ \texttt{Test}\downarrow \rangle$ adds an auxiliary node with label 1 to the graph and checks whether any unmarked nodes remain, and $\texttt{Forget}\!\downarrow$ removes all marks. The rules of $\texttt{Connected?}_2$ are shown in Figure 6.

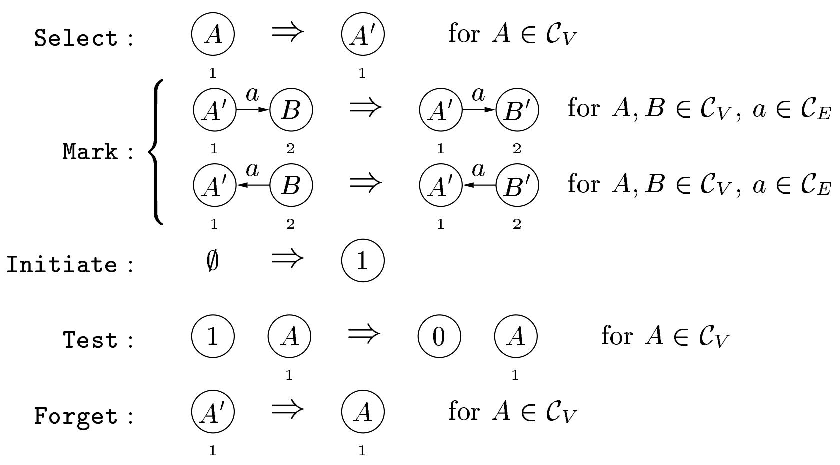


Fig. 6. The rules of `Connected?`$_2$

3.5 Testing for acyclicity

The function acyclic?: $\mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}'}$ with

$$\text{acyclic?}(G) = \begin{cases} G + 1 & \text{if } G \text{ is acyclic} \\ G + 0 & \text{otherwise} \end{cases}$$

is computed by the following program:

$$\texttt{Acyclic?} = \langle \texttt{Copy};\ \texttt{Reduce};\ \texttt{Check};\ \texttt{GarColl}\downarrow \rangle.$$

The idea behind this program is that a graph is acyclic if and only if the rules of Figure 7 reduce it to a graph without edges. The program `Reduce` for reducing the copy of the input graph is given by

$$\texttt{Reduce} = \langle \texttt{MarkNonLeaf}\downarrow;\ \texttt{DeleteEdge};\ \texttt{DeleteEdge}\downarrow;\ \texttt{Restore}\downarrow \rangle \downarrow,$$

with rules as shown in Figure 7. `Reduce` first marks all nodes with at least one outgoing edge, so that all other nodes must be leaves (nodes without outgoing edges). Edges pointing to leaves cannot belong to cycles and hence can be safely removed. This may result in new leaves, so we remove all marks and start again. The reduction finishes if no edges pointing to leaves remain. Note that the twofold occurrence of `DeleteEdge` guarantees that `Reduce` terminates.

The program `Check` = ⟨`Add`; `Test`↓⟩ adds an auxiliary node with label 1 to the graph, checks whether the reduced copy contains an edge and, if so, changes the label of the auxiliary node to 0. Finally, `GarColl`↓ removes the remainder of the copy. See Figure 8 for the rules of `Check` and `GarColl`.

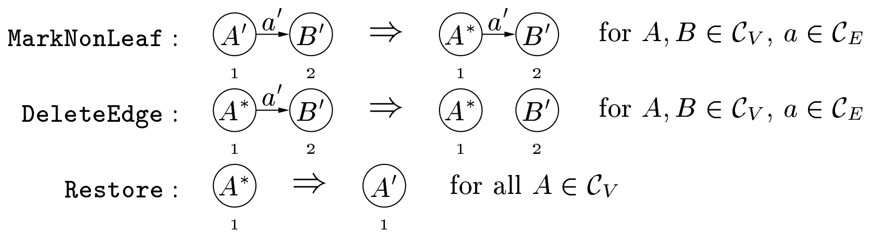


Fig. 7. The rules of `Reduce`

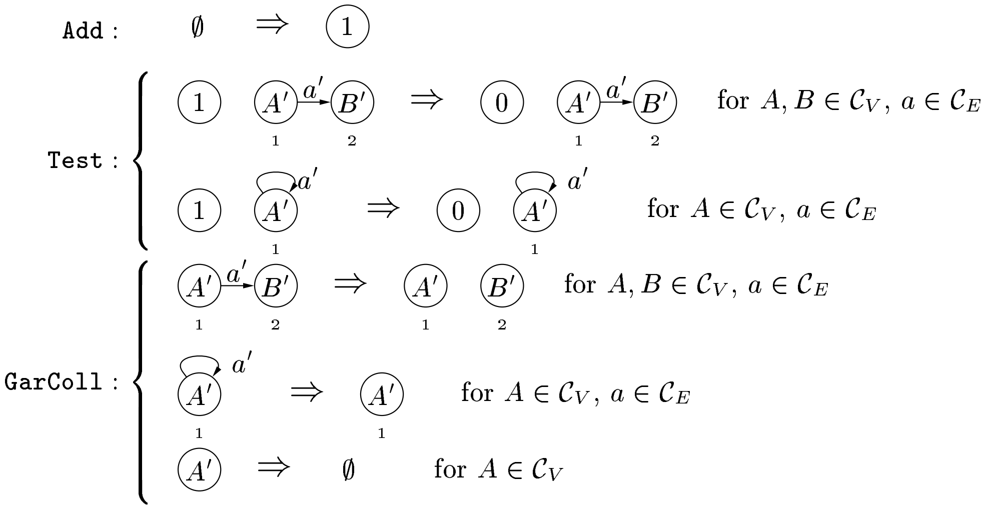


Fig. 8. The rules of `Check` and `GarColl`

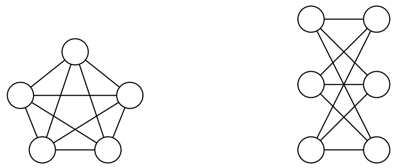
3.6 Testing for planarity

A graph is planar if it can be drawn on the plane without edge crossings. Our program for computing the function planar?: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_{\mathcal{C}'}$ with

$$\text{planar?}(G) = \begin{cases} G + ① & \text{if } G \text{ is planar,} \\ G + ⓪ & \text{otherwise} \end{cases}$$

is based on Kuratowski's Theorem.[2] By this theorem, an undirected and unlabelled graph is planar if and only if it has no subgraph homeomorphic to K_5 or $K_{3,3}$ [5]. Here K_5 is the complete graph with five nodes, and $K_{3,3}$ is the complete bipartite graph whose node sets both have 3 nodes, see Figure 9. Furthermore,

[2] It is known that this leads to an algorithm of exponential complexity, which is evident for our solution as `PowerGraph` produces a graph of exponential size. But simulating one of the subtle linear algorithms for planarity testing (see for example [1]) is beyond the scope of this paper.



K_5 $K_{3,3}$

Fig. 9. The graphs K_5 and $K_{3,3}$

two graphs are homeomorphic if both can be obtained from the same graph by a sequence of edge subdivisions. It follows that a graph is planar if and only if no subgraph reduces by repeated applications of the rule `Contract` in Figure 10 to a graph containing K_5 or $K_{3,3}$.

The function planar? is computed by the program

$$\texttt{Planar?} = \langle \texttt{Copy};\ \texttt{Simplify}{\downarrow};\ \texttt{PowerGraph};\ \texttt{Contract}{\downarrow};\ \texttt{Check};\ \texttt{GarColl} \rangle$$

whose rules are shown in Figure 10. The subprogram `Simplify`$\downarrow$ first transforms the copy of the input graph to an undirected, unlabelled graph without multiple edges and loops. We draw unlabelled nodes as unfilled circles, and undirected edges as lines without arrowheads. Implicitly, unlabelled nodes and edges carry a special "invisible"label and undirected edges are pairs of edges pointing in opposite directions.

The program `PowerGraph` of Section 3.2 is used to generate the disjoint union of all subgraphs of the copied input graph (where we only consider subgraphs resulting from edge deletions). Then `Contract`$\downarrow$ contracts the obtained graph as long as possible, and

$$\texttt{Check} = \langle \texttt{Initiate};\ \texttt{Test}(K_5){\downarrow};\ \texttt{Test}(K_{3,3}){\downarrow} \rangle$$

checks whether the contracted graph contains K_5 or $K_{3,3}$. The program first adds an auxiliary node with label 1 which, if the check was successful, is changed to 0. The interfaces of the rules $\texttt{Test}(K_5)$ and $\texttt{Test}(K_{3,3})$ consist of K_5 and $K_{3,3}$, respectively. Finally, `GarColl` removes the remainder of the simple graph.

A Appendix

A.1 The program scheme $\underline{\text{if}}$ _ $\underline{\text{then}}$ _ $\underline{\text{else}}$ _

We use a program scheme $\underline{\text{if}}\ K\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2$ which checks whether the input graph equals K and executes P_1 or P_2 depending on whether the check is successful or not. More precisely, the semantics is given by $G \to_{\underline{\text{if}}\ K\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2} H$ if and only if $G = K$ and $G \to_{P_1} H$ or $G \neq K$ and $G \to_{P_2} H$. The scheme is defined by

$$\underline{\text{if}}\ K\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2 = \langle \texttt{Check}(K);\ \langle \texttt{Delete}_1;\ P_1 \rangle{\downarrow};\ \langle \texttt{Delete}_2;\ P_2 \rangle{\downarrow} \rangle,$$

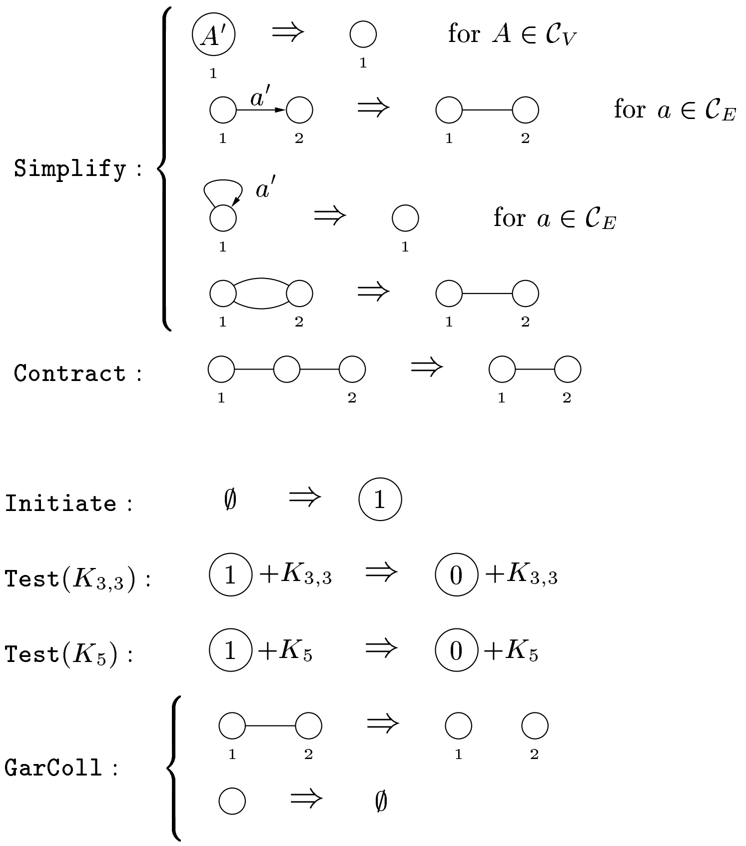


Fig. 10. The rules of Planar?

where Check(K) copies the input graph G and reduces the copy to a node with label 1 if $G = K$, and to a node with label 2 otherwise. For $i = 1, 2$, Delete$_i$ deletes a node with label i. If Check(K) yields 1, then $\langle$Delete$_1$; $P_1\rangle$ can be executed only once because the node with label 1 is deleted and $\langle$Delete$_2$; $P_2\rangle$ is executed zero times because there is no node with label 2. Vice versa, if Check(K) yields a node with label 2, then $\langle$Delete$_1$; $P_1\rangle$ is executed zero times and $\langle$Delete$_2$; $P_2\rangle$ is executed once. We omit the rules of this program scheme for space reasons.

A.2 The program scheme Copy

We also use a program scheme Copy for copying graphs. Given a label alphabet $C = \langle C_V, C_E\rangle$, let $C^{\copyright} = \langle C_V \cup (\mathcal{C}_V \times \{'\}) \cup (\mathcal{C}_V \times \{^*\}),\ C_E \cup (\mathcal{C}_E \times \{'\}) \cup (\mathcal{C}_E \times \{^*\})\rangle$. Labels $\langle l,' \rangle$ and $\langle l,^* \rangle$ from $C^{\copyright}$ are written l' and l^*, respectively. Copy transforms a graph G over C into the graph $G + G'$ over $C^{\copyright}$, where G' is obtained from G by replacing each label l with l'. Copy is defined as follows:

$$\texttt{Copy} = \langle \texttt{CopyNode}\downarrow;\ \texttt{CopyEdge}\downarrow;\ \texttt{Restore}\downarrow\rangle.$$

The rules of `Copy` are shown in Figure 11.

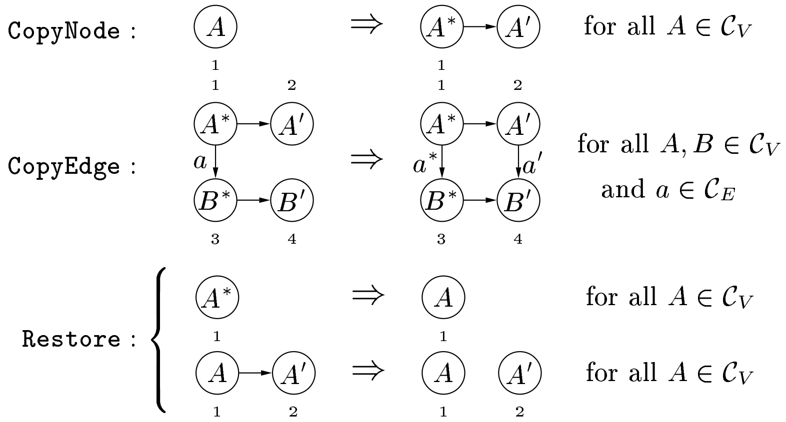


Fig. 11. The rules of `Copy`

References

1. Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.
2. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
3. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS '01)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.
4. Annegret Habel and Detlef Plump. Relabelling in graph transformation. In preparation, 2002.
5. Frank Harary. *Graph Theory.* Addison-Wesley, 1969.
6. Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11:690–723, 1999.

Author Index