European School on Graph Transformation

Bremen, March 2–7, 1998

Abstracts of Lectures

edited by Frank Drewes

Contents[1]

[1]In case of several authors lecturers are mentioned first.

[Lecture(s): Friday 11.00–12.30]

Pullback rewriting

Michel Bauderon
LaBRI, Université Bordeaux I
F-33405 - Talence (France)
Michel.Bauderon@labri.u-bordeaux.fr

After almost thirty years of graph rewriting, a large number of mechanisms have been introduced which generate various classes of graphs with different properties, using their own mechanism described in their own formalism making the comparison quite difficult and the necessity of a unifying framework clearer and clearer.

The basic difficulty comes from the fact that graphs can be considered from two rather different points of view, either as sets of vertices linked by edges or as sets of edges glued by vertices, each leading to a different kind of *item rewriting*, basically *edge rewriting* (or hyperedge replacement, shortly HR) and *node rewriting* (or vertex replacement, shortly VR).

In both cases, the basic ingredients of the mechanism are given by specifying what is to be replaced, how it is linked to the rest of the graph, by what it will be replaced and how the replacing part will be connected to the remaining part of the original graph. The main difference between both types of rewriting is probably that node rewriting may create new edges in an unpredictable way, while edge rewriting does not actually create, but simply unites already existing items into a single object.

This makes a big difference when trying to develop a more abstract setting for graph rewriting. Indeed, considering the graph to be rewritten as "embedded" in the big graph, it quickly appeared that the categorical generalisations of union and equivalence relation, namely coproduct and coequalizer where enough to give a good description of edge-oriented rewriting giving rise to the well known *double-pushout* approach. Unfortunately, this approach cannot describe the creation of an arbitrary number of edges as necessary during the step of context embedding and thus is applicable to the formalisation of node rewriting systems.

Other points of view have been developed since, providing a universal algebra approach and an extensive logical theory of graphs. The table in figure 1 summarises the mains characteristics of both types of graph rewriting together with the main developments achieved.

To fill the none in the table, it turns out that one has to use a dual point of view, going from embedding to projection, from union to product and from pushout to pullback. Further investigation of this new framework shows that it is much richer than expected, being able to encode most types of graph rewriting known so far, appearing as a good candidate for a universal generating mechanism.

The notion of a *structured graph* is introduced to gather in a common framework all kinds of graphs or hypergraphs. The underlying idea is that all graphs considered in the

	Hyperedge Replacement	Vertex replacement
Nature of the graphs	set of edges glued by nodes	set of nodes linked by edges
Substitution	replaces an edge	replaces a node
Interface	a family of nodes	a set of edges
Connection	glues interface nodes	creates interface edges
Algebraic framework	Available	Available
Categorical approach	Pushout rewriting	- - - None - - -
Logical Theory	Available	Available

Table 1: Hyperedge vs. Vertex replacement

literature are enriched versions of *ordinary graphs* given as sets of vertices and edges, without further detail or adornment.

The enrichment can be seen as a partitioning of the set of nodes in several subsets which will differ by their behaviour or more precisely by the way we shall interpret them. The most frequent case is probably that of hypergraphs seen as bipartite graphs, but in the very same way, one may describe entity-relationship diagrams, Petri nets or graphs with a more complicated structure. Structuring is done by a morphism to an appropriate graph (called the *structure graph*) having as many nodes as we need different kinds of behaviours for our nodes and whose edges will describe the potential relationship between all those kinds of nodes.

An *alphabet* is traditionally a set of labels used to describe the behaviour of a specific item of an object (letter in a word, node or edge in a graph, ...) during the rewriting. To describe a more complicated situation where this behaviour depends on other items in the object (neighbours for instance), a mere set of labels is no longer sufficient and some more information is needed such as the connection relation used in the classical works on node rewriting. To gather all necessary notions in a single structure, we use a structured graph and describe by a single morphism the behaviour of an item (of any kind), of its neighbourhood (with an appropriate notion of a neighbourhood) and of all context items (by definition those items which will not be affected by the rewriting).

The basic notions - unknown, rewriting rule, occurrence - are then appropriate arrows and the rewriting mechanism is defined as the product of an unknown and a rule (or their pullback depending on the category which one considers).

[Lecture(s): Monday 14.30–16.00, Tuesday 11.45–12.30]

Algebraic approaches

Hartmut Ehrig
TU Berlin
Theoretische Informatik TFS
Sekr. FR6-1, Franklinstr. 28/29
D-10587 Berlin (Germany)
ehrig@cs.tu-berlin.de

Andrea Corradini & Reiko Heckel
Dipartimento di Informatica
Università di Pisa
Corso Italia, 40
I-56125 Pisa (Italy)
{andrea,reiko}@di.unipi.it

This group of lessons will introduce the basics of the algebraic approach to graph transformation, and more specifically of the Double-Pushout (DPO) approach.

As an introduction we will start with a brief history of the algebraic approaches, and with a survey of the main application areas.

Next we introduce the basic notions of the double pushout approach, like graph, production and derivation. We explain how the intuitive idea of rule-based graph transformation is formalized by the double-pushout construction. In particular, we will discuss the correspondence between Graph Transformation Systems and Petri Nets. The basic idea is to regard Petri nets simply as graph transformation systems that act on a restricted kind of graphs, namely discrete, labeled graphs (that can be considered as sets of tokens labeled by places). Therefore graph transformation systems generalize Petri nets because they allow for arbitrary (also non-discrete) graphs, and also because they allow for the specification of context-dependent operations, where part of the state is read but not consumed. From this point of view, we will argue that the double-pushout construction is a natural generalization of the basic notion of "firing" of a net transition.

We then present some classical notions and results, including parallel independence of direct derivations, Parallelism Theorem, shift equivalence, and canonical derivations. Such concepts, which are the core of the theory of concurrency for algebraic graph transformations, are shown to be related to corresponding notions of Petri Nets and Term Rewriting.

Following we survey some other classical concepts that are useful for the modeling of concurrent and distributed systems. These include amalgamation and sequential composition of productions for constructing complex productions out of simpler ones, and distributed graphs and derivations for describing distributed systems.

We recall then some more recent extensions of the algebraic DPO approach, including consistency conditions for graphs and application conditions for graph productions, and structuring concepts for graph transformation systems.

Finally, we discuss the relationship between the DPO approach and the more general Single-Pushout (SPO) approach, which is able to model deletion in unknown contexts, stressing, for example, the different properties that they enjoy concerning the interleaving of parallel derivations.

All along the lectures a toy example based on the PacMan game is used.

[Lecture(s): Tuesday 11.00–11.45, Wednesday 11.00–11.45, Thursday 9.00–9.45]

The logical expression of graph properties and graph transformations

Bruno Courcelle
LaBRI, Université Bordeaux I
F-33405 - Talence (France)
courcell@labri.u-bordeaux.fr

The purposes of the lectures are twofold:

- first to give an overview of the chapter: "The expression of graph properties and graph transformations in monadic second-order logic" of the "Handbook of graph grammars and computing by graph transformations" and introduce the reader to the technical material,
- second to discuss related definitions, to show the intrinsic difficulties behind them, to indicate some blind alleys to the researcher interested to develop the theory.

1. Introduction

The logical expression of graph properties and graph transformations is linked to several fields of computer science:

- in descriptive complexity, one tries to "capture" complexity classes by logical languages: one has for instance a logical characterization of NP, but finding one of P is still open,
- in the theory of context-free graph grammars, monadic second-order logic is an essential tool, like are finite automata and rational transductions for the study of context-free languages,
- linear algorithms can be constructed in systematic ways for testing certain graph properties expressible in monadic second-order logic,
- in the theory of relational databases, logic is useful to express queries and integrity constraints (the state of a data base in this model is nothing but a hypergraph),
- in graph theory, certain still unknown finite sets of forbidden configurations can in principle be effectively constructed by means of logical tools (although in practice the methods are intractable),
- in the theory of graph drawings one can also hope for applications of logical and grammatical techniques.

2. Logical languages

Definitions will be given concerning the use of first-order logic, first-order logic enriched with least-fixed point operators, second-order logic, monadic second-order logic for expressing graph properties. Examples of expressible and nonexpressible properties will be given.

3. Logic and graph transformation

In classical model theory, transformations of logical structures expressed by logical formulas are called "interpretations" and are used to reduce a decision problem for a theory to one for another theory, and to prove new (un)decidability results from old ones. It is thus essential that a logical property P of the resulting logical structure be expressed in the given logical structure by a formula obtained in a uniform way from P and the formulas expressing the considered transformation of logical structures. Let us call this compatibility. Many useful graph transformations (spanning trees, connected components, quotients, etc…) can be expressed by monadic second-order formulas and are thus compatible in this sense with monadic second-order logic.

4. Graph operations and context-free graph grammars

Context-free graph grammars can be investigated in terms of systems of recursive equations (defining sets of finite graphs), written with set union and appropriate graph operations, typically gluings or relabellings. The operations on which the context-free graph grammars discussed in the basic core lectures on node-rewriting and hyperedge replacements are based, can be described in terms of disjoint union of graphs and quantifier-free graph transformations. The basic results showing compatibility between monadic second-order logic and context-free graph grammars follow quite easily. Follow also easily the applications to graph algorithms.

5. Extensions and open questions

One can enrich the notion of a graph into that of a map. A map is a graph given with a local planar embedding of the edges incident with each vertex. This is ap propriate to formalize (partially) drawings of planar graphs without edge crossings. The extension to the formalization of graph drawings with edge crossings will be discussed. Other recent results and open problems will be surveyed.

[Lecture(s): Thursday 16.30–18.00]

High-level replacement systems with parallelism and applications to algebraic specifications and petri nets

Hartmut Ehrig & Magdalena Gajewsky
TU Berlin
Theoretische Informatik TFS
Sekr. FR6-1, Franklinstr. 28/29
D-10587 Berlin (Germany)
{ehrig,gajewsky}@cs.tu-berlin.de

Francesco Parisi-Presicce
Università di Roma *La Sapienza*
Dipartimento di Scienze dell'Informazione
Via Salaria, 113
I-00198 Roma (Italy)
parisi@dsi.uniroma1.it

The general idea of High-level replacement Systems is to generalize the concept of graph transformation systems and graph grammars from graphs to all kinds of structures which are of interest in Computer Science and Mathematics. Within the algebraic approach of graph transformation this is possible by replacing graphs, graph morphisms and pushouts (gluing) of graphs by objects morphisms and pushouts in a suitable category. Of special interest are categories for all kinds of labelled and typed graphs, hypergraphs, algebraic specifications and Petri nets.

In this couple of lectures we show how main results for graph transformation systems concerning parallelism and horizontal structuring can be reformulated in the framework of high-level replacement systems.

The first main application shows how high-level replacement systems can be instantiated by algebraic specifications. An algebraic transformation rule corresponds to the interface part of an algebraic module specification for software systems. This allows to apply high-level replacement techniques to software system design. As an application it is shown how to reuse an algebraic module specification of an airport schedule for the design of a book library.

The second main application shows how rule-based modification of Petri nets can be considered as special case of high-level replacement techniques. An important result is the compatibility of horizontal structuring of nets with rule-based modification. This result is essential within a case study of a medical information system where the functional essence is developed by rule-based modification from the actual state of the system represented by algebraic high-level nets.

Finally we present a summary of important results within the theory of high-level replacement systems and their main application areas.

[Lecture(s): Monday 11.00–11.45, Tuesday 9.00–9.45, Wednesday 9.00–9.45]

Node replacement graph grammars

Joost Engelfriet
Department of Computer Science
Leiden University
P.O. Box 9512
NL-2300 RA Leiden (The Netherlands)
engelfri@wi.leidenuniv.nl

The aim of these three basic lectures is to present an introduction to the theory of context-free graph grammars that are based on node replacement. In the case of node replacement, as opposed to edge replacement, a node that is labeled with a nonterminal symbol (in a given graph) is removed and replaced by a graph which is the right-hand side of a production of the grammar. The embedding mechanism of the production connects that graph to the remainder of the given graph. Thus, the context-free graph grammar generalizes the usual context-free grammar on strings. It generates a graph language (i.e., a set of graphs) in a recursive fashion, and hence models a recursive property of graphs. As opposed to the case of strings, there are many natural types of context-free graph grammars. Here we discuss the most powerful context-free node replacement grammar, called (for historical reasons) the C-edNCE grammar. To stress its prominent role among the node replacement grammars, it is also called the NR grammar or the VR grammar (where V stands for vertex). The most prominent type of context-free graph grammar that is based on edge replacement, is called the HR grammar (where H stands for hyperedge).

These lectures are based on Chapter 1 of the Handbook. As far as time permits, the following topics will be treated. In the first lecture we give examples of node replacement grammars and explain the formal definition of NR (or C-edNCE) grammar. Then we discuss a number of basic concepts, such as: graphs with embedding, graph substitution, associativity of substitution, and blocking edges. To guarantee context-freeness of the NR grammar, it has to be required that the result of a derivation does not depend on the order in which the productions are applied (which is called confluence of substitution). In the second lecture we discuss some more basic concepts such as left-most derivations and derivation trees of NR grammars. These generalize the corresponding concepts for ordinary context-free grammars. Then we consider several natural subclasses of the class of the NR grammars and their expressive power, including a class that has the same power as the HR grammars. Normal forms of NR grammars (such as Chomsky normal form, and neighbourhood preserving normal form) will be discussed briefly. In the last lecture we consider several alternative characterizations of the class of graph languages generated by NR grammars, showing the robustness of the class. These are characterizations by: regular (tree) languages, logic, handle replacement, and sets of expressions.

[Lecture(s): Thursday 11.00–12.30]

Modularization

Gregor Engels
Fachbereich Informatik
Universität-GH Paderborn
Warburgerstr. 100
D-33098 Paderborn (Germany)
engels@uni-paderborn.de

A challenging issue of software engineering is the question how to master the complexity of large software systems. Currently, a variety of approaches try to solve certain aspects of this problem. One of the oldest approaches is to modularize large system specifications. This enables to hide details within a module, to reuse modules within different contexts, or to develop a system in a distributed work environment. In particular, to improve the reuse of modules, application-domain dependent frameworks have been investigated and built in the last decade. Frameworks are interconnected sets of modules, which provide a core functionality for a certain domain and which can be customized by refining their functionality. Last, view-oriented or aspect-oriented approaches are heavily discussed in the software engineering world. They support the basic principle of "separation of concern" and allow to build up a system from several parts, where each part describes or realizes only one aspect of the complete system.

The talk gives an overview on current approaches to incorporate those software engineering techniques into specification languages which are based on graph transformation systems. In particular, several modularization and view-oriented approaches are discussed in detail.

[Lecture(s): Monday 11.45–12.30, Tuesday 9.45–10.30, Wednesday 9.45–10.30]

Hyperedge replacement

Annegret Habel
Universität Hildesheim
Institut für Informatik
Marienburger Platz 22
D-31141 Hildesheim (Germany)
habel@informatik.uni-hildesheim.de

Abstract

In this lecture the concept of hyperedge replacement is presented as an elementary approach to graph and hypergraph generation. In particular, hyperedge replacement graph grammars are discussed as a (hyper)graph-grammatical counterpart to context-free string grammars. To cover a large part of the theory of hyperedge replacement, structural properties and decision problems, including the membership problem, are addressed.

Introduction

Graph grammars and graph languages have been studied since the late sixties and have been motivated by various application areas such as pattern recognition, semantics of programming languages, compiler description, data base systems, specification of data types, developmental biology, etc. The general approaches cover lots of potential applications; the more specific approaches, node rewriting and edge replacement, promise a rich mathematical theory.

In this lecture, we propose a kind of compromise by introducing hyperedge-replacement grammars reviving some ideas from the early seventies which can be found in Feder's and Pavlidis' work. On the one hand, Petri nets, functional expressions, program flow diagrams, molecule structures, and many other complex information structures can be interpreted in a natural way as hypergraphs (the hyperedges of which may have many "tentacles" attached to source and target nodes rather than only two "arms" like the edges of ordinary graphs). On the other hand, the replacement of hyperedges provides a type of context-free rewriting on the level of hypergraphs saving many of the fundamental properties of context-free rewriting on strings and ordinary graphs. Hence, we can be confident that this framework combines the theoretical attractivity of a class of context-free grammars and languages with the prospect for nice applications.

The lecture will concentrate on the following topics:

- Introduction
- Hyperedge replacement grammars
- A context-freeness lemma
- Structural properties
- Desision Problems

- The membership problem

The lecture will follow the handbook articles of Drewes et al. and Engelfriet ([DHK97, Eng97]), and volume 643 of Lecture Notes in Computer Science ([Hab92]).

References

[DHK97] Frank Drewes, Annegret Habel, Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Grzegorz Rozenberg, ed., Handbook of Graph Grammars and Computing by Graph Transformations, volume 1: Foundations, 95–162. World Scientific, Singapore, 1997.

[Eng97] Joost Engelfriet. Context-free graph grammars. In Grzegorz Rozenberg, Arto Salomaa, eds., Handbook of Formal Languages, volume 3: Beyond Words, 125–213. Springer-Verlag, Berlin, 1997.

[Hab92] Annegret Habel. Hyperedge Replacement: Grammars and Languages, volume 643 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.

[Lecture(s): Thursday 14.30–16.00]

Graph-rewriting semantics for concurrent systems

Dirk Janssens
Department of Mathematics and Computer Science - WISINF
University of Antwerp - UIA
Universiteitsplein 1
B-2610 Wilrijk (Belgium)
dmjans@uia.ua.ac.be

The two lectures provide an overview of recent work concerning the use of embedding-based GTS in concurrency. The main aim is to show that one may set up a simple and flexible framework for the definition of a wide class of GTS, then develop a true concurrent (process) semantics for that general framework, and finally specialize to GTS suitable for the particular application one may be interested in. Two examples of such specializations are considered: one leads to a rule-based semantics for an actor language and the other one concerns the verification of a simple protocol. It is also shown that the issue of compositionality can be dealt with at the general level, and the relevance of the general notions and results with respect of the two applications is demonstrated.

Firstly, the general ideas of a rule-based semantics are presented, and their potential advantages are discussed. Then two possible application areas are sketched: on the one hand, it is shown that the dynamic behaviour of an actor system can be captured by a GTS of a specific type, called Actor Grammars, and it is investigated which properties this GTS should have if it is to be used as the basis for an implementation on multiprocessor machine. On the other hand, a GTS description of a simple transport protocol is given, and it is discussed how the correctness of the protocol can be expressed.

Next the general framework is intoduced. The basic idea is that productions are considered as molecular operations built from atomic node replacements; these are very simple operations replacing a single node by another node, while computing the neighbourhood of the new node on the basis of that of the replaced node (the neighbourhood of a node is its label together with the labels of the edges adjacent to it). Building productions happens by gluing together node replacements; this gluing can be viewed as a form of synchronization.

By choosing the sets of node- and edge-labels, and the node replacement operations acing on them, one may specialize the general model to obtain concrete models. It is shown that one can get Petri nets, Actor Grammars and ESM systems in this way. Furthermore, it turns out that each GTS obtained from the general framework allows a process semantics (in the sense of Petri net pocesses) if one assumes a simple condition on the algebraic structure of node replacement operations. This process semantics can be made compositional by defining a suitable composition operation on processes, sets of processes and systems. The composition operation on processes is very general in the sense that it encompasses both the sequential and the parallel composition of processes but it does require a refinement of the original notion of a process.

Finally, the two applications sketched earlier are considered in more detail: it is shown how an actor language with a traditional syntax can be translated into a set of GTS productions, and a compositional proof of the transport protocol is developed.

Contents

[Lecture(s): Saturday 11.45–12.30]

Picture generation (by collage grammars)

Hans-Jörg Kreowski
Fachbereich Mathematik und Informatik
Universität Bremen
Postfach 33 04 40
D-28334 Bremen (Germany)
kreo@informatik.uni-bremen.de

As picture generation is one of the central issues of computer graphics and grammars are well-known and successful devices for the generation of objects of various kinds, one encounters quite a variety of syntactic approaches to the generation of artificial pictures and classes of pictures in the literature. Collage grammars belong to this variety and are used in this survey in order to explain some of the principles and possibilities of picture generation. They show some specific characteristics that seem to be worth mentioning.

(1) Collage grammars allow one to derive collages from collages, that is, they work directly on pictorial objects rather than on representations of some kind that need an extra pictorial interpretation.

(2) The derivation process of collage grammars employs the concept of hyperedge replacement which stems from the field of graph grammars. Hence, the study of collage grammars can be seen as an attempt to carry over some experience in graph grammar theory to computer graphics.

(3) As hyperedge replacement is a context-free mode of rewriting, the theory of collage grammars presents various results with counterparts in the theory of context-free Chomsky grammars and of Lindenmayer systems.

A collage consists of a set of parts and a sequence of pin-points. A part may be an arbitrary set of points (in a Euclidean space)—usually taken from some standard set of geometric objects like line segments, circles, triangles, polygons, polyhedra, etc. that have simple finite descriptions and are easy to deal with on graphical surfaces. The pin-points are used to paste collages into collages. A collage represents a picture by the overlay of its parts. To generate collages from collages by the application of production rules, they are decorated with hyperedges in intermediate steps. A hyperedge is a labelled item with an ordered number of tentacles each of which is attached to a point. A hyperedge serves as a place holder. It may eventually be replaced by a decorated collage, provided that there is a transformation of the pin-points into the attachment points of the hyperedge. This kind of hyperedge replacement establishes the rewrite steps of a collage grammar if the label of the replaced hyperedge and the replacing decorated collage form a production. So, a collage grammar generates a set of collages in the usual way of language generation. As collages represent pictures, a set of pictures is generated at the same time. The overlay of all generated pictures and their intersection yields two particular pictures that are called the upper and lower generated fractal.

[Lecture(s): Thursday 9.45–10.30, Friday 17.15–18.00]

Graph relabelling systems and distributed computations

Yves Métivier
LaBRI, Université Bordeaux I
F-33405 - Talence (France)
Yves.Metivier@labri.u-bordeaux.fr

Graph relabelling systems have been introduced as a suitable model for expressing and studying distributed algorithms on a network of communicating processors. We recall the basic ideas underlying that model and we survey the main questions that have been considered and the main results that have been obtained in that framework.

Contents

[Lecture(s): Friday 9.00–10.30]

Concurrent semantics for graph rewriting[1]

Ugo Montanari
Dipartimento di Informatica
Università di Pisa
Corso Italia, 40
I-56125 Pisa (Italy)
ugo@di.unipi.it

Graph rewriting systems are the generalization of string grammars to graphs. Besides of generating graph languages, they can be used for the specification of concurrent and distributed systems. In this respect, they are far more expressive than, for example, Petri nets. In fact, they allow for a finer description of a state (which is a graph, instead of a marking, which is a set or a multiset), and for the possibility of specifying context-dependent transitions (that is, transitions which consume only some of the preconditions needed to apply them). It is therefore potentially very useful to develop a model of computation and a semantics of graph rewriting in the style of other concurrent formalisms, like Petri nets or process description languages.

A model of computation for a graph rewriting system is a mathematical structure which contains relevant information concerning the potential behaviour of the rewriting system, that is, about all the possible computations (derivations) of the system. Since the basic operation on derivations is concatenation, it is quite obvious that categories are well suited as mathematical structures for this use. In fact, a model of computation for a given grammar is just a category having graphs as objects, and where every arrow is a derivation starting from the source graph and ending at the target graph. Then the categorical operation of sequential composition of arrows corresponds to the concatenation of derivations.

In previous lectures, graph rewriting systems according to the double-pushout variant of the algebraic approach have been introduced, together with their categories of derivations [1]. In the most abstract model, the so-called truly-concurrent model, the arrows of the category actually correspond to equivalence classes of derivations, where isomorphic (in a very precise sense) and shift-equivalent computations have been merged. Two computations are shift equivalent if they differ only for the order in which some independent, concurrent rewritings are actually performed. Thus the arrows of this category correspond to the possible, abstract observations performed on a reactive system by an external, asynchronous agent. Notice however that these arrows do not only form a set, but they keep the information about the nondeterministic branching structure, since the fact that in general a graph has several rewritings possible is represented by several outgoing arrows from the corresponding node.

[1]Research partly supported by the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems) and by the EC Esprit WG APPLIGRAPH (Applications of Graph Transformation).

In the talk we show how it is possible to unfold the truly-concurrent model into an event structure, or, equivalently, into a prime algebraic domain [2]. These models are the most well-studied semantic domains for concurrent systems, and are equipped with general, important constructions and properties. Intuitively, an event structure identifies two transition occurrences which differ only for agents idle in the transition, but distinguishes all the events which have different past histories. The talk aims at giving an intuitive understanding of these events, via some representation theorems. To this purpose, it is useful to introduce graph processes [3], a generalization of Goltz-Reisig processes for place-transition Petri nets. As Goltz-Reisig processes represent deterministic computations of nets, and are themselves acyclic nets, similarly graph processes are actually acyclic graph rewriting system. More precisely, Goltz-Reisig processes are essentially graph processes for the special case of graph rewriting systems, where all the involved graphs have no arcs and all the rewriting rules have no contexts. More generally, there is a direct correspondence of the theory we develop for graph rewriting and the concurrent theory of nets.

To precisely state our representation theorems, it is necessary to introduce slightly modified versions of graph processes [4], where the source, and, possibly, the target graphs are suitably decorated to avoid endomorphisms different from the identity. Then we show that the arrows of the truly-concurrent model, the events of the event structure and the elements of the prime domain can all be characterized as certain graph processes. These representation results are useful not only to understand the semantics of graph rewriting systems in itself, but also to capture the right intuitions when graph rewriting systems are used to represent other models of computations.

References

[1] Corradini, A., Ehrig, H., Lwe, M., Montanari, U. and Rossi, F., Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach, in: G Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, pp. 163-245.

[2] Corradini, A., Ehrig, H., Loewe, M., Montanari, U. and Rossi, F., An Event Structure Semantics for Graph Grammars with Parallel Productions, in: J. Cuny, H. Ehrig, G. Engels and G. Rozenberg, Eds., *Graph Grammars and Their Application to Computer Science*, Springer LNCS 1073, pp. 240-256.

[3] Corradini, A., Montanari, U. and Rossi, F., Graph Processes, *Fundamenta Informaticae*, vol. 26, n. 3/4 (1996) pp. 241-265.

[4] Baldan, P., Corradini, A. and Montanari, U., Concatenable Graph Processes: Relating Processes and Derivation Traces, *paper submitted for publication.*

[Lecture(s): Friday 14.30–16.00]

Modeling concurrent, mobile and coordinated systems via graph rewriting[1]

Ugo Montanari
Dipartimento di Informatica
Università di Pisa
Corso Italia, 40
I-56125 Pisa (Italy)
ugo@di.unipi.it

In this talk we model three rather different formalisms for concurrent systems, taking advantage of the operational machinery of double pushout graph rewriting systems and of their concurrent semantics as developed in previous lectures. The three formalisms are: i) concurrent constraint programming, ii) π-calculus, and iii) grammars for distributed systems. They have all been well described in the literature, and two of them, i) and ii), are actually the most studied formalisms in their respective fields.

The concurrent constraint logic programming framework [1, 2] extends both logic programming and concurrent logic programming in that a program consists of the concurrent execution of agents which add (i.e., "tell") and check (i.e., "ask") constraints on a shared set of variables, and whose behaviour is described by a set of clauses. This formulation is very general and can be seen as a concurrent logic programming shell which is parametrized w.r.t. the underlying constraint system. Graphs and graph rewriting systems can be conveniently used to describe such a framework [1], and the modeling is so elegant and expressive that they provide what we believe is the most natural abstract machine for concurrent constraint programming. All the operational and denotational semantics for the concurrent constraint programming paradigm proposed in the literature treat the shared store as a monolith and thus are not useful for deriving any information about the causal dependencies among the agents or the maximal degree of parallelism. On the contrary, the semantics we propose, which is based on the concurrent semantics of graph rewriting systems, is able to formally express these concepts. In [2], contextual Petri processes, an extension of Petri processes with transitions consuming only some of the preconditions needed to apply them, are used instead of graph processes for the sake of simplicity, but it is easy to move to the more descriptive graph processes when needed.

The π-calculus is one of the best studied examples of mobile process calculi, namely calculi in which the communication topology among processes can dynamically evolve when computation progresses. The π-calculus extends the CCS calculus in that it allows for the communication of channel names. It can also be used to model higher-order systems and object-oriented systems, since names can be seen as objects, or higher-order

[1]Research partly supported by the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems) and by the EC Esprit WG APPLIGRAPH (Applications of Graph Transformation).

continuations. While the π-calculus can be naturally applied to distributed systems, its ordinary formal semantics is not concurrent and does not refer to any notion of locality or causality. In the talk, following [3], we give instead a (both operational and abstract) concurrent semantics for the π-calculus mapping agents into certain finite graph rewriting systems, and exploiting their concurrent semantics according to the double pushout approach. For the abstract semantics we introduce interleaving, partial ordering and mixed ordering observations, define the corresponding bisimulation relations and compare them with those proposed in the literature.

The third formalism we consider are grammars for distributed systems (GDS) [1]. Here we assume that a process, in order to evolve, may require to synchronize with adjacent processes. That is, if two processes share a piece of data, then each of them may pose some conditions on this shared piece of information. If they agree on these conditions, then both of them (but not one of them alone) can evolve. Thus the evolution of a single process may happen only if the process is isolated, or if it does not impose any synchronization condition on the adjacent processes. A possible simple way to model process evolution in such kind of systems is to use a two-phase approach: first context-free process productions are specified (a set for each process), as well as synchronization requests for each of these possible moves. Then, context-sensitive rewriting rules are obtained by combining compatible context-free productions. These context-sensitive rules allow for the evolution of a subpart of the system consisting of several processes that agree on conditions on the shared data. Applying the rule means making all such processes (and not a proper subset of them) evolve. This constraint restricts the way a system evolves, and it provides a natural formalism to model the behaviour of distributed systems with decentralized control. In fact, once the processes have agreed on how to behave on the shared data, they may move without any need of communicating with each other: each process needs only to know its selected context-free production. This approach models a simple version of process coordination, where in general the interaction of a coordinator with ordinary agents (or other coordinators) may involve more complex synchronous or asynchronous steps. In the talk, following [5], we illustrate the coordination understanding of GDS and hint on how the actual computational problem can be efficiently solved via constraint programming techniques.

References

[1] Montanari, U. and Rossi, F., True-Concurrency in Concurrent Constraint Programming, in V. Saraswat and K. Ueda, Eds., *Proc. International Logic Programming Symposium '91*, MIT Press, pp. 694-713.

[2] Montanari, U. and Rossi, F., Concurrency and Concurrent Constraint Programming, in: Andreas Podelski, Ed., *Constraint Programming: Basics and Trends*, Springer LNCS 910, 1995, pp. 171-192.

[3] Montanari, U. and Pistore, M., Concurrent Semantics for the π-calculus, in: S. Brookes, M. Main, A. Melton and M. Mislove, Eds., *Procs. MFPS '95*, ENTCS, pp. 337-356.

[4] Degano, P. and Montanari, U. A Model of Distributed Systems Based on Graph Rewriting, *Journal of the ACM* Vol. 34, N2, April 1987, pp. 411-449.

[5] Montanari, U., and Rossi, F., Graph Rewriting and Constraint Solving for Modelling Distributed Systems with Synchronization, in: Paolo Ciancarini and Chris Hankin, Eds., *Coordination Languages and Models*, LNCS 1061, 1996, pp. 12-27. Full paper to appear in *Applied Category Theory.*

[Lecture(s): Tuesday 15.15–16.00, Wednesday 11.45–12.30]

Term graph rewriting in functional programming

Rinus Plasmeijer
Department of Computer Science
Toernooiveld 1
NL-6525 ED Nijmegen (The Netherlands)
rinus@cs.kun.nl

Quoted from the preface of the book *Term Graph Rewriting - Theory and Practice* (Wiley, 1993):

> *Term graph rewriting* refers to techniques and theories for representing terms and term rewrite rules as graphs and graph rewrite rules. Many modern programming paradigms - most notably the functional and logic paradigms - have term rewriting at their heart, although the control regimes and semantics are very different. Practical implementations of these languages share subterms using pointers, and the machine code for such programs manipulates shared structures.

[Lecture(s): Monday 16.30–17.15, Tuesday 14.30–15.15]

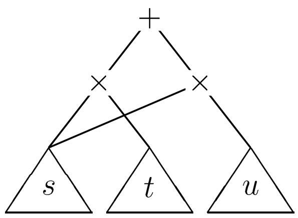
Term graph rewriting

Detlef Plump
Fachbereich Mathematik und Informatik
Universität Bremen
Postfach 33 04 40
D-28334 Bremen (Germany)
det@informatik.uni-bremen.de

Equations over functional expressions are a prevalent concept in formal specification and declarative programming. Sets of equations serve as specifications and programs, and computing with equations includes reasoning about specifications and evaluating expressions. In both cases, equations are employed as directed rewrite rules which replace subterms.

In implementations of rewriting, one usually avoids to represent expressions as strings or trees because this may enforce a copying of subexpressions which multiplies evaluation work. For example, the application of a distributive law $\mathtt{x} \times (\mathtt{y} + \mathtt{z}) = (\mathtt{x} \times \mathtt{y}) + (\mathtt{x} \times \mathtt{z})$ to a term of the form $s \times (t + u)$ duplicates the subterm s which subsequently has to be evaluated twice.

This kind of inefficiency can be overcome when expressions are represented by graph-like structures which allow to *share* common subexpressions. For instance, the term $(s \times t) + (s \times u)$ resulting from the just mentioned rewrite step can be compactly represented by the following graph structure (where the triangles stand for representations of the subterms s, t, and u):



With this representation, the subterm s has to be evaluated only once. To put it in another way, an evaluation of s in the graph corresponds to a parallel evaluation of two occurrences of s in the string or tree representation.

The name *term graph rewriting* has come into use for the evaluation of expressions by graph transformation. In this lecture, we explain how term graph rewriting works and present fundamental properties of this computational model. In particular, we highlight the relations to the conventional term rewriting model. The lecture will be structured as follows:

1. Introduction
2. Term graphs
3. Term graph rewriting
4. Soundness and completeness
5. Termination
6. Confluence
7. Modularity
8. Term graph narrowing
9. Further topics

[Lecture(s): Friday 16.30–17.15, Saturday 11.00–11.45]

2-structures

Grzegorz Rozenberg
Department of Computer Science
Leiden University
P.O. Box 9512
NL-2300 RA Leiden (The Netherlands)
rozenber@wi.leidenuniv.nl

The notion of 2-structure is more general than the notion of graph but less general than the notion of relational structure. These structures can be used to study various combinatorial properties of systems such as graphs, partially ordered sets and communication networks, where all kinds of (binary) relationships hold between the objects of a system.

The theory of 2-structures covers both the "static" part of the theory concerned mainly with hierarchical representations and the "dynamic" part concerned with local transformation.

In our lecture we give a tutorial introduction to the theory of 2-structures.

[Lecture(s): Tuesday 16.30–18.00]

Software development with programmed graph rewriting systems

Andy Schürr
Lehrstuhl für Informatik III
Aachen University of Technology (RWTH)
Ahornstr. 55
D-52054 Aachen (Germany)
andy@i3.informatik.rwth-aachen.de

Graphs play an important role within many areas of applied computer science, and there exists an abundance of visual languages and environments which have graphs as their underlying data model. This includes even object-oriented software engineering modeling languages such as UML, if we do accept that "objects" and "nodes" as well as "assocations" and "edges" are just different names for the same concepts.

Rule-based languages and systems, on the other hand, are well-suited for the description of complex transformation or inference processes on complex data structures. Why not combining two successful principles in one formalism? This is the common vision of the graph grammar research community, which is now about 30 years old. Furthermore, it was the main motivation behind the development of PROGRES, a very high-level language, which supports PROgramming with Graph REwriting Systems.

The language PROGRES has a graph-oriented data model and a graphical syntax for its most important language constructs. It was developed having the following design goals in mind:

- Distinguish between data definition and manipulation as database programming languages do and use graph scheme definitions to typecheck graph rewrite rules.
- Do not rely on the rule-oriented programming paradigm for all purposes, but support also imperative programming with usual control structures.
- Use a graphical syntax for graph rewrite rules as well as for graph schemes, but use a textual syntax for attribute conditions and assigments or control structures.
- Refrain users from the task to guarantee confluence of defined rewriting systems by keeping track of rewriting conflicts and backtracking out of dead-end derivations.

The PROGRES programming environment is the result of ten years of work. It offers assistance for creating, analyzing, compiling, and debugging graph rewriting system specifications. Being an integrated set of tools with support for intertwining these activities, PROGRES combines the flexibility of interpreted languages with the safeness of compiled and statically typed languages.

The PROGRES environment is available as free software for Sun workstations via ftp from `ftp-i3.informatik.rwth-aachen.de` in the directory `/pub/PROGRES` or from `ftp.qucis.queensu.ca` in the directory `/pub/PROGRES` or via its world-wide web home page `http://www-i3.informatik.rwth-aachen.de/research/progres/index.html.`

It is nowadays used at various sites around the world for specification and rapid prototyping activities:

- Collegues at the University of Technology Aachen need PROGRES for rapid prototyping configuration management and process modeling tools of integrated software engineering and chemical process engineering environments.
- Software engineers from the University of Leiden and TNO Delft have been using PROGRES for the specification of visual (database query) languages and for prototyping a process modelling tool.
- Researchers from the University of Paderborn are prototyping database reverse engineering applications with PROGRES.
- Finally, there are activities at institutions like Queen's University, Kingston, INRIA, Sophia Antopolis, and Carleton University, Ottawa, where PROGRES is used for pattern recognition of mathematical expressions, for manipulating conceptual graphs, and for analysing distributed system traces, respectively.

[Lecture(s): Monday 17.15–18.00]

Foundations of the PROGRES approach

Andy Schürr
Lehrstuhl für Informatik III
Aachen University of Technology (RWTH)
Ahornstr. 55
D-52054 Aachen (Germany)
andy@i3.informatik.rwth-aachen.de

Various forms of programmed graph rewriting systems as extensions of context-sensitive graph rewriting systems have been proposed until today. They differ considerably with respect to their underlying graph models, the supported forms of graph rewrite rules, and offered rule regulation mechanisms. Some of them have additional constructs for the definition of graph schemes, derived graph properties, and so forth. The graph rewriting language PROGRES is one example of this kind. It combines rather complex forms of graph rewrite rules and imperative control structures for the definition of graph transformations with object-oriented class diagrams for the definition of graph types and directed equations as well as first-order logic expressions for the definition of derived attributes and static integrity constraints.

It was a rather difficult and time-consuming task to develop a precise and complete, but nevertheless compact formal definition of the language PROGRES. Programmed Logic-based Structure Rewriting (PLSR) systems were developed as a kind of intermediate definition language for this purpose.

- They treat specific graph classes as sets of predicate logic formulas with certain properties, so-called structures.
- Their rules preserve the consistency of manipulated structures and use nonmonotonic reasoning for checking needed pre- and postconditions.
- So-called Basic Control Flow (BCF) expressions together with an underlying fixpoint theory provide needed means for programming with rules.

This talk introduces the basic framework of PLSR systems and shows by means of examples, how PROGRES specifications are formally defined by translating them into PLSR systems.

[Lecture(s): Saturday 9.00–10.30]

Distributed graph transformation with application to visual design of distributed systems

Ingrid Fischer & Victor Volle
University of Erlangen-Nürnberg
Department of Computer Science (IMMD)
Martenstraße 3
D-91058 Erlangen (Germany)
{idfische,vrvolle}@informatik-uni-erlangen.de

Gabriele Taentzer & Manuel Koch
TU Berlin
Theoretische Informatik TFS
Sekr. FR6-1, Franklinstr. 28/29
D-10587 Berlin (Germany)
{gabi,carr}@cs.tu-berlin.de

Distributed systems demand a number of requirements on specification techniques which do not have to be taken into account for the development of non-distributed software. Allocation of objects and tasks, object replication and migration, remote interactions, multiple threads of control as well as dynamic network topologies are important issues in distributed systems. Some of these distribution issues are already handled by common modeling techniques based on graphical notations, but altogether distributed systems cannot be designed sufficiently with these techniques.

In this lecture, new concepts for visual design of distributed systems are presented which are based on distributed graph transformation as underlying formal specitfication technique. The kernel of this technique focuses on a structured graph with two abstraction levels, the network and the local level, and its transformation. On the network level, the (possibly) dynamic topological structure of a distributed system is specified. The local level contains the graphical description of local object and data structures where parts of them may be replicated in remote network nodes. Those structures may evolve independently or synchronized with remote structures using a suitable interaction mechanism. Network activities as well as local actions may be described rule-based. The advantage of using rules in this context is to avoid purely sequential or serie-parallel control flow, since rules naturally support the design of multiple threads of control on the level of events without pre! scribing the order of execution. The graphical representation of network as well as data structures uses a notation following that of UML, a recently developed object-oriented modeling language.

Distributed graph transformation is based on the double-pushout approach to graph transformation. Network as well as local graph transformations may be attributed and may use application conditions for rules. Using graph transformation as underlying formal framework, distributed behaviour is designed in a way that consistency of the network as well as of all object and data structures involved is ensured. This results from the formalization of distributed graph transformation as double-pushout in the category of distributed graphs and graph morphisms.

For the illustration of the concepts presented we use a reference application modeling distributed version control for any kind of document. This subject becomes the more important the more remote software development becomes popular. In contrast to server-client solutions where all documents are held centrally, the documents are really distributed over the involved sites in our case.

To round this lecture, distributed graph transformation is compared with other graph transformation approaches used to model distributed systems. Furthermore, this approach is roughly compared with other specification and modeling techniques for distributed systems.