URL: http://www.elsevier.nl/locate/entcs/volume2.html 9 pages

Encapsulated Hierarchical Graphs, Graph Types, and Meta Types

Gregor Engels [a], Andy Schürr [b]

[a] *Department of Computer Science, Leiden University*
P.O. Box 9512, NL-2300 RA Leiden, The Netherlands

[b] *Lehrstuhl für Informatik III, RWTH Aachen*
Ahornstr. 55, D-52074 Aachen, Germany

Abstract

Currently existing graph grammar-based specification languages have serious problems with supporting any kind of "specification-in-the-large" activities. More precisely, they have deficiences with respect to modeling hierarchical data structures or specifying meta activities like manipulation of graph schemata. Furthermore, already proposed graph grammar module concepts are still too abstract to be useful in practice. Our contribution addresses these problems by introducing a new hierarchical graph data model with an infinite number of schema, meta-schema, etc. layers. It forms the base for a forthcoming concrete modular graph grammar specification language where in addition information hiding aspects like explicit export and import interfaces are expressible.

1 Introduction

After 25 years of research, graph grammars and graph transformation systems have reached a certain degree of maturity. They were and are succesfully used for writing rather complex specifications of software engineering tools [4], visual database query language [1], and the like. Nevertheless, our experiences with writing these specifications show that currently used graph data models and graph grammar specification languages have still serious deficiencies with respect to "programming in the large" activities (cf. [11]):

(i) It is unacceptable that all data of a specified complex system has to be modelled as a single flat graph. In contrast, a hierarchical graph data model would be useful, where certain details of "encapsulated" subgraphs may be hidden, but subgraph boundary crossing edges are still supported.

(ii) There are real needs for a graph grammar module concept with support for inheritance and genericity, such that big specifications may be constructed as assemblies of small reusable subspecifications with well-defined interfaces between them.

(iii) And even the already established idea of combining graph rewrite rules for specifying dynamic system aspects and graph schemata for specifying static system aspects needs further improvements. Additional means would be welcome for defining (meta-)schemata of graph schemata and for specifying even schema modifying operations.

First proposals addressing these problems do exist, but they are either on a very abstract level [3] or offer only partial solutions for modularization [8]. Furthermore, various papers are already published which deal with topic (i) above [2,6,7,9,13,14], the definition of a hierarchical graph data model. Unfortunately, these papers do not address the problem of information hiding or disallow even subgraph crossing edges.

Therefore, we felt the necessity to study first concepts centered around hierarchical graphs and graph types before being able to design a more concrete graph grammar module concept. The result of these studies - or more precisely - our attempts to transfer already known software engineering or database design concepts to the world of graphs and graph grammars is a formal definition of a hierarchical graph data model which solves the above mentioned problems. A subset of these definitions is presented over here. For a complete description of the new data model, containing all definitions as well as (additional) theorems with proofs and more detailed examples, the reader is referred to [5].

2 Hierarchical Graphs

In the sequel, we will introduce hierarchical graphs and the principle of information hiding by discussing the following example: We have a world of companies which know each other (or not) and which produce and trade articles. The whole situation is modelled as a hierarchical graph, where each company is a complex node, i.e. a graph in its own right. For instance, a big company consists of departments as well as of R&D departments, workers (within all kinds of departments), as well as various kinds of produced or consumed articles. Relationships between companies, like "know each other" or "negotiate about something" are represented by binary directed edges. This situation is illustrated in Fig. 1.

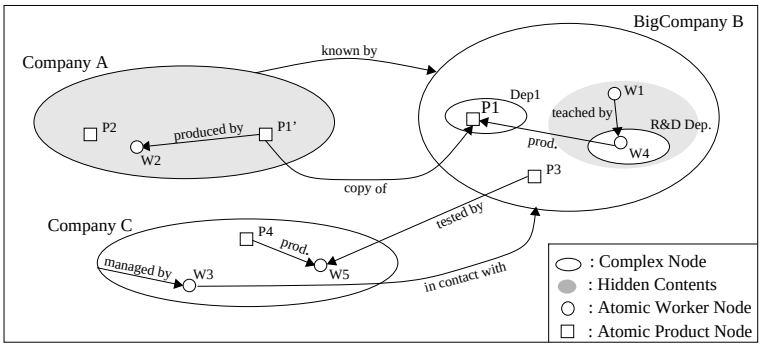

Fig. 1. A hierarchical graph with partially hidden graph contents

By employing the information hiding principle, the world of companies can be modelled more appropriately:

- Any complex node in a graph has its own private state. Such a state is a so-called enapsulated graph with protected or even invisible nodes and edges for other complex nodes within the same graph (e.g. the results of an R&D department of a company).
- Any complex node knows parts of its context in the surrounding graph and may even be the source or target of a graph boundary crossing edge (e.g. a product P1' of a company A may be a copy of another product P1 manufactured within a department of another company B).
- Furthermore, complex nodes may have access to visible components within known context nodes (an employee W5 of a company C is assigned to test a product P3 belonging to a company B, which is a visible fact for both companies).

The formal definition of such a hierarchical graph model with support for information hiding is split into two parts: the introduction of *encapsulated graphs* with support for information hiding and the definition of *hierarchical graphs* themselves. It is based on the definition of a set of **atomic nodes** $\mathcal{N}$ which have a unique **identifier** oid and a **type** label tid. Any atomic node belongs to a type layer $\mathcal{A}_i$ with $\mathcal{A}_0$ being the set of all ordinary nodes, $\mathcal{A}_1$ the set of their types etc. The *type* of a node n of a layer $\mathcal{A}_i$ is a node t of layer $\mathcal{A}_{i+1}$ such that: $oid(t) = tid(n)$. Furthermore, a set of edge labels $\mathcal{E}$ is needed.

Definition 2.1 (Encapsulated Graphs)
A tuple $G := (KN, H, HN, KE, E, HE)$ is an **encapsulated graph** over a given set of nodes $\mathcal{N}$, $G \in \mathcal{G}(\mathcal{N})$, with:

(i) $KN(G) := KN \subseteq \mathcal{N}$ is the set of *known* nodes in G.

(ii) $N(G) := N \subseteq KN$ is the set of all nodes which *belong* to G.

(iii) $HN(G) := HN \subseteq N$ is the set of all *hidden* nodes in G.

(iv) $KE(G) := KE \subseteq KN \times \mathcal{E} \times KN$ is the set of *known* binary edges in G.

(v) $E(G) := E \subseteq KE$ is the set of all edges which *belong* to G.

(vi) $HE(G) := HE \subseteq E$ is the set of all *hidden* edges in G.

Definition 2.2 (Abbreviations)
The following abbrevations are needed in the sequel with G being a graph:

(i) $CN(G) := KN(G) \setminus N(G)$ is the set of all *context* nodes in G.

(ii) $CE(G) := KE(G) \setminus E(G)$ is the set of all *context* edges in G.

(iii) $VN(G) = N(G) \setminus HN(G)$ is the set of all *visible* nodes in G.

(iv) $VE(G) = E(G) \setminus HE(G)$ is the set of all *visible* edges in G.

Definition 2.3 (Hierarchical Graphs and Complex Nodes)
A triple $c := (oid, G, tid)$ is a **complex node** of layer k, $c \in \mathcal{C}_k$, which contains a **hierarchical graph** G of layer k, $G \in \mathcal{G}(\mathcal{C}_k)$, if and only if it belongs to the smallest set of elements which fulfill the following conditions:

(i) $oid(c) := oid$ is a unique node identifier and $tid(c) := tid$ is the identifier of the complex node $t \in \mathcal{C}_{k+1}$, denoted as $type(c)$, such that $tid(c) = oid(type(c))$.

(ii) $G(c) := G \in \mathcal{G}(\mathcal{C}_k)$ is the node's internal state, a hierarchical graph of layer k (a flat graph if containing atomic nodes only).

(iii) $\forall n \in \mathcal{A}_k \subseteq \mathcal{C}_k : G(n) := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ is the empty flat encapsulated graph, i.e., the set of atomic nodes $\mathcal{A}_k$ is embedded in the set of complex nodes $\mathcal{C}_k$.

(iv) $\forall n \in N(G(c)) : CN(G(n)) \subseteq KN(G(c)) \wedge CE(G(n)) \subseteq KE(G(c))$, i.e. all known elements in n not belonging to n must be known in c.

(v) $\forall n \in N(G(c)) : VN(G(n)) \subseteq N(G(c)) \wedge VE(G(n)) \subseteq E(G(c))$, i.e. all visible elements of subnodes of c belong to c, too.

(vi) $\forall n \in HN(G(c)) : VN(G(n)) \subseteq HN(G(c)) \wedge VE(G(n)) \subseteq HE(G(c))$, i.e. all visible elements of hidden subnodes of c belong to c as hidden elements.

(vii) $\forall n \in N(G(c)) : KE(G(c)) \cap (KN(G(n)) \times \mathcal{E} \times KN(G(n))) \subseteq KE(G(n))$, i.e. all known edges in c, whose sources and targets are known in a subnode n of c, are known in n, too.

The definition above of hierarchical graphs is necessarily *recursive*. A complex node contains a hierarchical graph as its state, which contains in turn atomic as well as complex nodes. Its requirements (i) to (iii) are straightforward. The remaining more complex conditions guarantee that knowledge about context elements and especially about *visible graph boundary crossing edges* is propagated up and down the hierarchy of nested graphs as needed.

Based on the definitions above *use* and *contains* relations as well as a *refines* (*subtype*) relation may be defined between nodes (node types). They are valid on the instance level as well as on the type or all meta-type i levels.

Definition 2.4 (Uses and Contains Relation)
Let $c, n \in \mathcal{C}_k$ be two nodes of layer k.

(i) c **uses** $n :\Leftrightarrow n \in KN(G(c)) \setminus N(G(c))$.

(ii) c **contains** $n :\Leftrightarrow n \in N(G(c))$.

The universe of all complex nodes is restricted by the additional condition that the contains relation builds a forest or more precisely a set of finite trees.

Assuming a properly defined refinement relation on atomic nodes we are able to define a similar relation on complex nodes:

Definition 2.5 (Refinement Relation)
A node c' **refines** another node c (with $c, c' \in \mathcal{C}_k$) of the same layer if:

(i) $(type(c') = type(c)) \vee (type(c')\ \textit{refines}\ type(c))$, i.e. the type of a refined node is either the same as before or a refined version in turn.

(ii) $\exists$ injective function $f : KN(G(c)) \rightarrow KN(G(c'))$, i.e. any node in c has to be mapped onto a unique node in its refinement c'.

(iii) $\forall n \in CN(G(c)) : f(n) = n \wedge f(n) \in CN(G(c'))$, i.e. refinement of a node does not include refinement of its context elements (the known context may be extended).

(iv) $\forall n \in N(G(c)) : f(n)$ *refines* $n \wedge f(n) \in N(G(c'))$, i.e. nodes within c may be replaced by refined versions of themselves within c'.

(v) $\forall n \in VN(G(c)) : f(n) \in VN(G(c'))$, i.e. refinement preserves visibility of nodes.

(vi) $\forall (n_1, e, n_2) \in CE(G(c)) : (f(n_1), e, f(n_2)) \in CE(G(c'))$ i.e. all context edges have to be preserved.

(vii) $\forall (n_1, e, n_2) \in E(G(c)) : (f(n_1), e, f(n_2)) \in E(G(c'))$, i.e. own edges have to be preserved, too.

(viii) $\forall (n_1, e, n_2) \in VE(G(c)) : (f(n_1), e, f(n_2)) \in VE(G(c'))$, i.e. even visibility of edges has to be preserved.

Please note that refinement is a rather general concept. It includes the usual definitions of *inheritance and subtyping* as they may be found in literature. It is complicated by the fact that we have recursion w.r.t. the contains hierarchy as well as w.r.t. to the hierarchy of type layers. Its soundness is shown by the following theorem (its proof is part of the technical report [5]).

Proposition 2.6 (Soundness of Refinement Relation)
The refinement relation is reflexive and transitive. It may be used to define a proper equivalence (isomorphy) relation between hierarchical graphs.

3 Hierarchical Graph Types and Meta Types

Up to now, we have illustrated the usage of hierarchical graphs to model real-world situations. All those graphs are typed and have to be instances of a corresponding graph type. As it is common to use Entity-Relationship diagrams, i.e., special forms of graphs, to define conceptual database schemes, we reuse our notion of hierarchical graphs to define a graph type. Consequently, this hierarchical graph (of a higher layer) is called *graph schema* and it defines instances, i.e. hierarchical graphs on an instance layer. Such a graph schema contains complex nodes as the definitions of *complex node types* and atomic nodes as the definitions of *atomic node types*. Edges between type nodes of a graph schema represent the permission to create edge instances between node instances of corresponding types. An example is given in Fig. 2, where the graph schema contains an edge "Company - known_by $\rightarrow$ Company". This declaration allows to create edges between two Company nodes.

Definition 3.1 A hierarchical graph S of layer $k+1$ serves as a **schema** for another graph G of layer k if:

(i) $\forall n \in KN(G)\ \exists t \in KN(S) : type(n)$ *refines* t, i.e. any known node has a node type which is at least a refinement of a known type within the corresponding schema (partial knowledge about outside world).

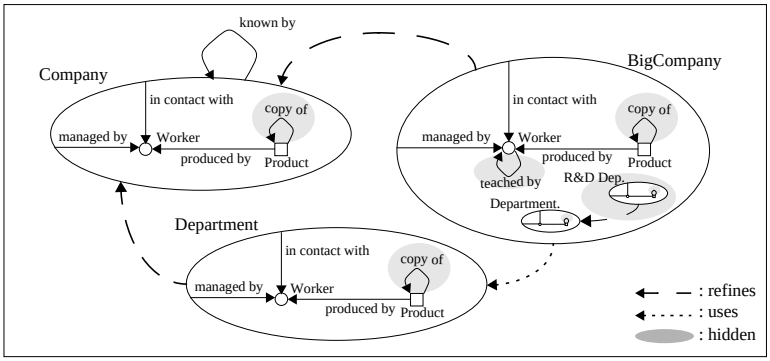

Fig. 2. A hierarchical graph schema with partially hidden contents

(ii) $\forall n \in N(G)\, \exists t \in N(S) : type(n) = t$, i.e. nodes belonging to G must have types belonging to S.

(iii) $\forall n \in VN(G)\, \exists t \in VN(S) : type(n) = t$, i.e. visible nodes may not have invisible type definitions in S.

(iv) $\forall (n_1, e, n_2) \in KE(G)\, \exists (t_1, e, t_2) \in KE(S)$:
$type(n_1)$ *refines* $t_1 \wedge type(n_2)$ *refines* t_2, i.e. any known edge must have a declaration in S such that its actual source and target types are refinements of the given types in S.

(v) $\forall (n_1, e, n_2) \in VE(G)\, \exists (t_1, e, t_2) \in VE(S)$:
$type(n_1)$ *refines* $t_1 \wedge type(n_2)$ *refines* t_2, i.e. a visible edge of G should have a visible definition in S.

In order to increase the understandability as well as reusability of graph schemata, we introduce additional means to structure a graph schema. First, complex nodes within a graph schema form *subschemata* as they define hierarchical graphs as instances of this complex node type. Second, two subschemata may be interrelated by two different kinds of interrelationships.

The *uses* relationship, well-known as module interrelation within the software engineering community, expresses that the definition of a complex node c' is imported by another complex node c to define the internal structure of c. For instance, in figure 2, the Department subschema is *imported* (used) by BigCompany to express that instances of Department nodes may occur within an instance of BigCompany.

The *refinement* or *inheritance* relationship, well-known as class interrelation within the object-oriented terminology, facilitates the definition of structurally similar objects. In this case, a graph schema is interrelated by a refinement relationship with another graph schema. This expresses the fact that the refining schema contains all types of the refined schema as well as additional and/or refined node and edge types. For example, Department is a refinement of Company, and R&D Department is a refinement of Department.

The BigCompany gives an example, where both the refinement and uses relationship are employed. BigCompany is a refinement of Company as it has Department nodes as additional components. Furthermore, it has a use relationship to Department for the same reason.

Finally, we have to emphasize that graph schemata are indeed nothing else but hierarchical graphs with a superimposed special interpretation. As a consequence, we are forced to deal with graph schemata of graph schemata, so-called *meta schemata*, and schemata of meta schemata. This leads to the construction of a type universe with an infinite number of layers. From a practical point of view, the first layer with ordinary graph objects and the second layer with their type objects (graph schemata) are the most interesting ones. The third layer of meta types (meta schemata) is important as soon as schema modifications or extensions have to be regarded.

4 Related Work and Summary

The presented graph data model has some similarities with the underlying data model of Hyperlog [13] and especially to the data model of Telos [10]. In Telos, both nodes and attribute values are modelled as so-called "tokens", and binary relationships between tokens play the role of edges over here. Furthermore, even our concept of an *infinite number of type layers* was influenced by the Telos concept of an infinite number of class layers. These layers do not only introduce classes of classes of ... of tokens, but allow even the definition of (meta) classes (categories) of binary relationships, a concept which is not supported over here. Up to now, we believe that edges should be kept as simple as possible. Otherwise, the dinstinction between nodes and edges vanishes step by step up to the point that n-ary edges between n-ary edges are allowed. In that case, new "connectors" have to be introduced which bind a given n-ary edge to all its corresponding partners. These connectors, sometimes also called "roles" (in ER data models), have then about the same purpose as primitive edges of the currently proposed data model and are, therefore, again second class objects without attributes, types, etc.

Nevertheless, there exists a number of graph data models which allow for *edges between edges* or which support *aggregation of edges*, like those of AGG [9] and EDGE [12]. There exists even the graph data model of Hy+ [2], which makes almost no distinction between ordinary edges and aggregation relationships as we do. As a consequence, this data model supports multiple simultaneously existing aggregation hierarchies, which makes the definition of suitable encapsulation or data abstraction concepts almost impossible.

To summarize, there exist quite different hierarchical graph data models in literature, with some of them being more than 15 years old [14]. Comparing them with the data model presented here, the main differences are that

- almost all of them — except Telos [10] and Hyperlog [13] — do not support a uniform treatment of graphs and graph schemata,
- many of them prohibit graph boundary crossing edges, and not a single one studies properties of graph boundary crossing edges in detail,
- and all but "pin graph" like models [7] neglect the importance of distinguishing between internal hidden nodes and externally visible nodes wich may be referenced by the outside world.

Finally, we have to admit that the presented data model is incomplete as long as *update operations* on encapsulated hierarchical graphs are not taken into account. These operations in the form of extended graph rewrite rules will manipulate hierarchical graph instances without violating the requirements mentioned in the definition 2.3 of hierarchical graphs and their accompanying schema integrity constraints. It is future work to define such a new kind of graph rewrite rules, define their semantics as binary relations over hierarchical graphs (as suggested in [3] and [8] for ordinary graph rewrite rules) and to extend the principal of refinement to graph rewrite rules. This means that we have to define *graph object types* which are hierarchical graph types (schemata) together with a set of schema consistency preserving rewrite rules. Refinement in this extended setting means then that a more specific graph object type may have additionally associated rewrite rules or rewrite rules with a refined definition, i.e. structural refinement (inheritance) presented over here is extended to behavioral refinement (inheritance).

Such a *refinement concept for rewrite rules* will be based on their semantics definition by means of binary relations and it will have about the following form: A graph rewrite rule r' refines a graph rewrite rule r if

$$\forall (G_1, G_2) \in \mathrm{Sem}(r')\ \exists (G_1', G_2') \in \mathrm{Sem}(r) : G_1 \text{ refines } G_1' \wedge G_2 \text{ refines } G_2'$$

It is our hope that the new data model of hierarchical graphs together with the sketched refinement concept for graph rewrite rules may be combined with a recently presented concept of graph transformations on distributed graphs, where different graphs share common subgraphs via so-called interface graphs [16]. A first attempt of incorporating the idea of information hiding into the framework of *distributed graph transformations* is subject of ongoing research activities and the subject of another submission to SEGRAGRA'95 [15].

References

[1] Andries M., Engels G.: Syntax and Semantics of Hybrid Database Languages, in: Ehrig H., Schneider H.-J.: Proc. Dagstuhl-Seminar 9301 on Graph Transformations in Computer Science, LNCS 776, Berlin: Springer Verlag (1994)

[2] Consens M., Mendelzon A.: Hy+: A Hygraph-based Query and Visualization System, in: Buneman P., Jajodia S. (eds.): Proc. 1993 ACM SIGMOD Conf. on Management of Data, SIGMOD RECORD, vol. 22, no. 2, acm Press (1993), 511-516

[3] Ehrig H., Engels G.: Towards a Module Concept for Graph Transformation Systems, Technical Report 93-34, Dept. of Computer Science, Leiden University, October 1993

[4] Engels G., Lewerentz C., Nagl M., Schäfer W., Schürr A.: Building Integrated Software Development Environments Part I: Tool Specification, in: acm Transactions on Software Engineering and Methodology, vol. 1, no. 2, New York: acm Press (1992), 135-167

[5] Engels G., Schürr A.: Encapsulated Hierarchical Graphs, Graph Types, and Meta Types (long version), Technical Report 95-21, Dept. of Computer Science, Leiden University, July 1995

[6] Himsolt M.: Hierarchical Graphs for Graph Grammars, in: Abstracts Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, 67-70

[7] Höfting F., Lengauer Th., Wanke E.: Processing of Hierarchically Defined Graphs and Graph Families, in: Monien B., Ottmann Th. (eds.): Data Structures and Efficient Algorithms, LNCS 594, Springer Verlag (1992), 45-69

[8] Kreowski H.J., Kuske S.: On the Interleaving Semantics of Transformation Units - A Step into GRACE, in: Abstracts Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, 400-405

[9] Löwe M., Beyer M.: AGG - An Implementation of Algebraic Graph Rewriting, in: Proc. 5th Int. Conf. on Rewriting Techniques and Applications, LNCS 690, Springer Verlag (1993), 451-456

[10] Mylopoulos J., Borgida A., Jarke M., Koubarakis M.: Telos: Representing Knowledge About Information Systems, in: ACM Transactions on Information Systems, vol. 8, no. 4, acm Press (1990), 325-362

[11] Nagl M., Schürr A.: Graph Grammar Specification Problems from Practice, in: Abstracts Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, 139-144

[12] Newbery Paulisch F.: The Design of an Extendible Graph Editor, LNCS 704, Springer Verlag (1993)

[13] Poulovassilis A., Levene M.: A Nested-Graph Model for the Representation and Manipulation of Complex Objects, in: ACM Transactions on Information Systems, vol. 12, no. 1, acm Press (1994) 35-68

[14] Pratt T.W.: Definition of Programming Language Semantics Using Grammars for Hierarchical Graphs, in: Claus V., Ehrig H., Rozenberg G. (eds.): Proc. Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology, LNCS 73, Springer Verlag (1979), 390-400

[15] Schürr A., Taentzer G.: DIEGO, another Step towards a Module Concept for Graph Transformations, Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, Volterra (Pisa), Italy, August 1995, Electronic Notes in Theoretical Computer Science (ENTCS), same vol., Elsevier Science Publ. (1995)

[16] Taentzer G.: Hierarchically Distributed Graph Transformations, in: Abstracts Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, 430-435