Technical Report 95–21 July 1995

Rijksuniversiteit te Leiden

Vakgroep Informatica

Encapsulated Hierarchical Graphs, Graph Types, and Meta Types

G. Engels

A. Schürr

Department of Computer Science
Leiden University
P.O. Box 9512
2300 RA Leiden
The Netherlands

Encapsulated Hierarchical Graphs, Graph Types, and Meta Types

Gregor Engels
Department of Computer Science
Leiden University, P.O. Box 9512
NL-2300 RA Leiden, The Netherlands
email: engels@wi.leidenuniv.nl

Andy Schürr
Lehrstuhl für Informatik III
RWTH Aachen, Ahornstr. 55
D-52074 Aachen, Germany
email: andy@i3.informatik.rwth-aachen.de

1. Introduction

After 25 years of research, graph grammars and graph transformation systems have reached a certain degree of maturity. They were and are successfully used for writing rather complex specifications of software engineering tools [5], visual database query languages [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. [15]):

(1) It is unacceptable that all data of a specified complex system have to be modelled as a single flat graph. In contrast, a *hierarchical graph data model* would be useful, where certain details of “encapsulated” (sub-)graphs may be hidden but subgraph boundary crossing edges are still supported.

(2) There are real needs for a graph grammar *module concept* with support for import/export relationships as well as for inheritance such that big specifications may be constructed as assemblies of small reusable subspecifications with well-defined interfaces between them.

(3) 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 or offer only partial solutions. Ehrig and Engels introduce in [4] an abstract framework for a *module concept*, which adapts the world of algebraic specification language module concepts to the world of graphs and graph transformations. Kreowski and Kuske present in [11] so-called graph *transformation units*, which are essentially groups of related rewrite rules together with certain graph class descriptions and rule controlling application conditions. Both papers address thereby topic (2) of our problem list above. Next, *two-level graph grammars* of Göttler [7] offer means to adapt a given graph grammar specification towards a more specific scenario and are thereby related to topic (3) above.

Furthermore, various papers [3, 8, 9, 12, 18, 17, 20, 22] are already published dealing with topic (1) above, 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 supports:

- *Encapsulation* (information hiding) as a means of hiding nodes and edges within graphs from the outside world.
- *Aggregation* as a means of defining hierarchical graphs, where each node may possess a complex state which is another graph.
- *Classification* as a means of defining static graph properties in the form of graph schemata, schemata of graph schemata, and … .
- *Refinement* (inheritance) as a means of using already existing schema definitions and extending them as needed for the description of more specific graph classes.

These concepts and their relationships will be studied within this paper. It has the following outline: Section 2 introduces a running example and explains needed concepts on an informal level. Section 3 afterwards offers all necessary formal definitions including some consistency proving theorems. Both sections discuss first encapsulation and aggregation, and proceed then with classification as well as refinement. Section 4 finally, compares our approach with related work and sketches how still missing concepts might be added later on, leading to a *graph grammar module concept for hierarchical and encapsulated graph objects*.

2. Informal Presentation of Hierarchical Graph and Types

2.1 Hierarchical Graphs and Complex Nodes

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 “normal” departments and R&D departments as well as of various kinds of produced or consumed articles. *Relationships* between companies as well as between workers and products within companies, like “know each other” or “is produced by” are represented by binary directed edges. This situation is illustrated in figure 1.

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 encapsulated graph whose nodes and edges may be declared as invisible for other complex nodes within the surrounding graph, e.g. the R&D department R&D_Dep1 in big company B with all its researchers is invisible for company A or C.

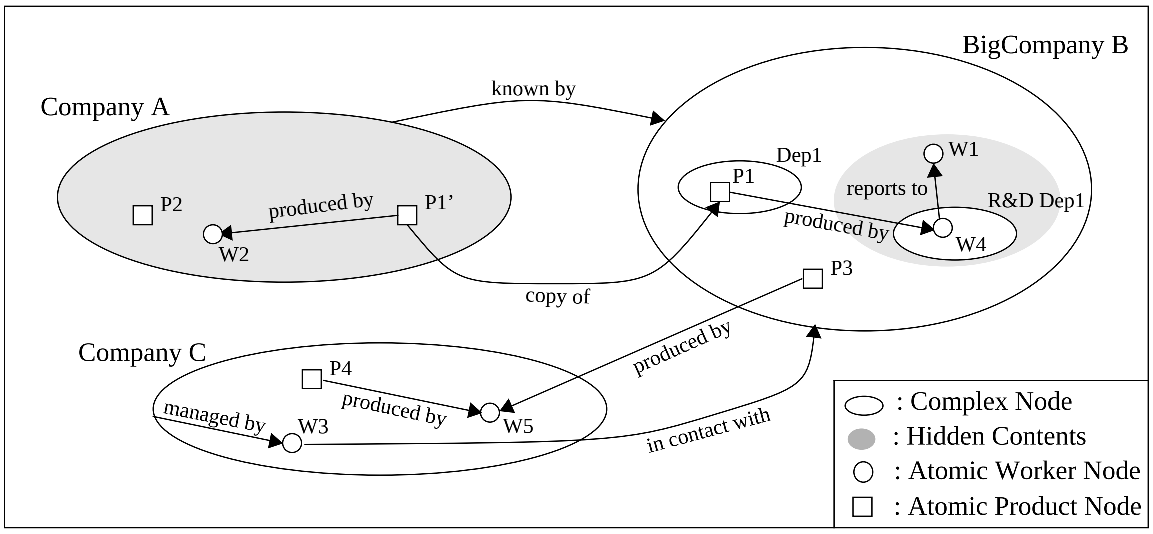

Fig. 1: A hierarchical graph with partially hidden graph contents.

- Any complex node knows parts of its context in the surrounding graph, i.e. the department A knows the atomic node P1in B, and it 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 of known context nodes (an employee W5 of a company C produced a product P3 which belongs now to another company B and which is a visible artifact for both companies).

The last point of the list above is the most important one and needs a more detailed explanation. Consider for instance the case of a company A which copies the products of a company B. Then, products of company A might be invisible for company B, but products of company B are visible for company A. The sample graph of Fig.1 has even a copy_of edge from A's product P1' to B's product P1. This edge is visible for the complex node A but invisible for the complex node B.

Figure 1 summarizes the overall situation. It shows a hierarchical graph with three company (sub-)graphs as its complex nodes. Company A has a completely hidden internal graph structure, whereas company B has a partially hidden structure (consisting of its R&D department and its worker W1). Company C, finally, reveals all its internal details to companies A and B. This is a simplified view of the capabilities of our data model. In the general case, companies A and B may have *knowledge about different potentially visible parts* of company C. This knowledge may even be expanded and contracted during their life times.

2.2 Hierarchical Graph Types and Meta Types

Up to now, we have illustrated the usage of hierarchical graphs to model real-world situations. All these graphs are typed and have to be instances of a corresponding *graph type*. As it is common to use Entity-Relationship like 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

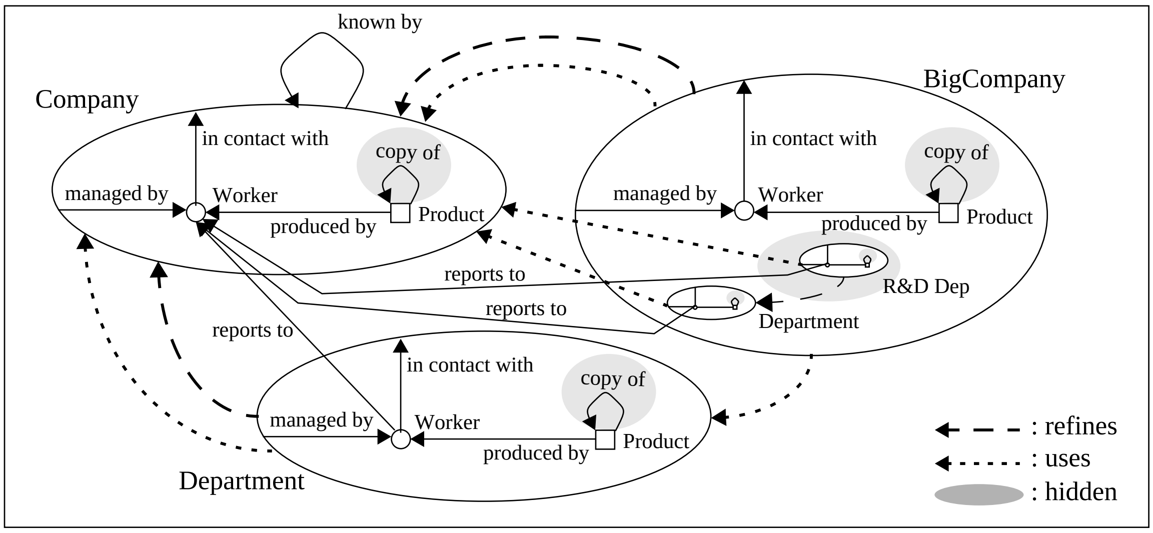

Fig. 2: A hierarchical graph schema with partially hidden contents.

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 figure 2, where the graph schema contains an edge "Company — known_by → Company". This declaration allows to create edges between two Company nodes.

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 of a hierarchical graph form *subschemata* as they define hierarchical graphs as instances of this complex node type. For example, the overall graph schema within Figure 2 contains the subschemata Company, BigCompany, and Department.

Furthermore, these subschemata contain in turn complex node types definitions (subschemata) as well as atomic node type definitions. Any contains relationship on the schema level represents possible contains relationships on the instance level. A BigCompany instance *may contain* an arbitrary number of Department instances and Department instances themselves *may contain* Workers, Products, … . The word "may" indicates that we do neglect any kind of cardinality constraints for reasons of simplicity, i.e. we are currently not able to express facts like "a BigCompany contains at least two Departments", "a Department contains one managed by edge and at least ten Workers", … .

Please note that the notion of a subschema is comparable to the notion of an *abstract data type module*, as known within the software engineering community, and to the notion of a *class*, as used in the object-oriented terminology. Consequently, we can and will reuse the structuring concepts from these two communities to interrelate subschemata.

Within the software engineering community, a module definition consists of an *export interface* defining the visible part of a module as well as of a hidden internal part. Two modules may be interrelated by a *uses* relationship. This means that there may be a link between an instance of the using module to an instance of the used module. The subschema Department uses for instance the Worker type definition of Company to define a reports_to relationship, which connects a Worker of a Department to a Worker of a Company (or a BigCompany or a Department, which are both refinements of Company; see below). As a consequence, Department and R&D_Dep1 within BigCompany as well as BigCompany itself need use relationships to Company for their (inherited) reports_to relationships. Another example of a use relationship exists between the subschemata BigCompany and Department. BigCompany uses the Department definition to express the fact that instances of BigCompany may contain instances of Department.

A structuring concept, well-known within the object-oriented terminology is the refinement or *inheritance* concept. It facilitates the definition of structurally and/or behaviorally similar objects. In this case, a graph schema is interrelated by a *refinement* relationship with another graph schema, expressing thereby the fact that the refining schema contains all type definitions of the refined schema as well as additional and or refined node and edge type definitions. For example, Department is a refinement of Company (it declares an additional reports_to edge) and R&D Dep1 is a refinement of Department in turn (it is a hidden declaration in BigCompany, whereas Department is a visible declaration).

Within the description above we did not make a clear distinction between Worker in Company and Worker in BigCompany or between the separate definition of Department and its usage in BigCompany. But we will see later on that it is necessary — from a theoretical point of view — to prefix all occurrences of atomic and complex node type definitions within a complex node type definition c with the name of c. Therefore, we have to distinguish between BigCompany.Department and Department itself or between BigCompany.Worker and Department.Worker or BigCompany.Department.Worker. A BigCompany.Worker, for instance, may or may not have additional properties in comparison to a simple Company.Worker, i.e. a refined complex node type (subtype) definition contains equivalent or refined copies of its supertype.

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.

3. Formal Definition of Hierarchical Graphs and Types

This section provides a formal definition of *schema consistent hierarchical graphs* without taking their accompanying graph transformations into account. Furthermore, it studies conceptual relationships between graphs, which may be divided into three main categories and are the subject of the following three subsections:

(1) A hierarchical graph, as for instance the hierarchical graph of section 1 (figure 1), *contains* nodes which are either atomic or complex. *Atomic nodes* are the leaves of our “contains” hierarchy and do not have an internal state of their own (attributes may be taken into account later on). *Complex nodes*, on the other hand, have an internal state, which is another hierarchical graph.

(2) Nodes, graphs, and later on node types and graph schemata may *refine* each other. A complex node (type) refines another complex node (type) if and only if its internal graph state is a refinement of the other node’s graph state.The definition of graph refinement itself is based on the definition of an injective functions between graphs.

(3) Finally, any node within a graph is an *instance* of another node of a “higher” layer, called node type. Node types of atomic nodes are atomic nodes themselves; node types of complex nodes are complex nodes, which have a graph schema as their internal state.

3.1 Hierarchical Graphs and Complex Nodes

This subsection introduces the basic data model of hierarchical graphs with visible and hidden nodes and edges. These graphs contain labeled atomic as well as complex nodes and labeled directed edges between them. Their type definitions are higher level graphs, which will be studied later on in subsection 3.2 and especially in 3.3.

The current version of the data model is based on three important *simplifications* which have to be withdrawn in the future:

- Atomic nodes and their node types are assumed to be defined elsewhere (together with appropriate refinement relationships between them).
- Attributes of nodes have to be modeled as atomic nodes within complex nodes.
- Edges are just ternary relations of the form (source id, edge label, target id), and they have neither an internal state nor a type declaration which may be refined.

The following two definitions provide us with the basic means for constructing graphs:

Def. 3.1 Basic Alphabets

In the sequel, we will need the following symbol sets:

(1) $\mathcal{NID}$ is a given set of **node identifiers**.

(2) $\mathcal{NL}$ is a given set of **node labels**.

(3) $\mathcal{EL}$ is a given set of **edge labels**. ❑

Def. 3.2 Atomic Nodes

A tuple n := (nid, nl) is an **atomic node**, in signs n ∈ $\mathcal{A}$, iff:

(1) *nid*(n) := nid ∈ $\mathcal{NID}$ is the node’s unique identifier.

(2) *nl*(n) := nl ∈ $\mathcal{NL}$ is the node’s label. ❑

Based on the definition of atomic nodes, we are now able to define flat encapsulated graphs with hidden nodes and edges. Afterwards, we will introduce hierarchical graphs themselves.

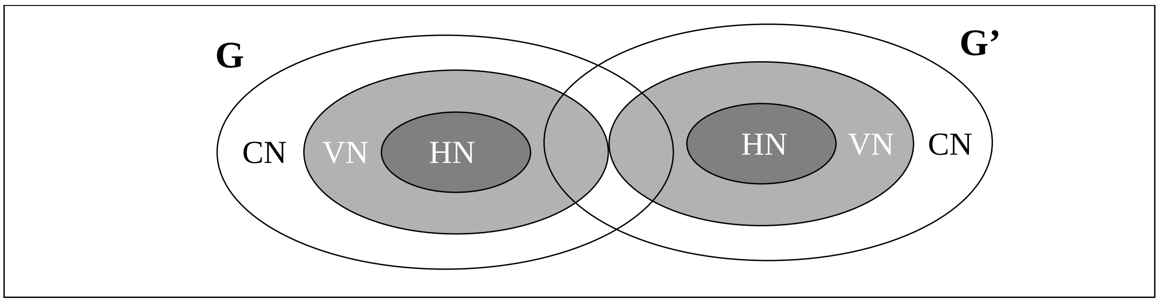

Fig. 3: Different kinds of nodes of two encapsulated graphs.

Def. 3.3 Encapsulated Graphs

A tuple G := (KN, N, HN, KE, E, HE) is an **encapsulated graph** over a given set of nodes $\mathcal{N}$ with visible and hidden nodes and edges as well as with knowledge about its context, in signs G ∈ $\mathcal{G}(\mathcal{N})$, iff:

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

(2) N(G) := N ⊆ KN is the set of all nodes, which belong to G.

(3) HN(G) := HN ⊆ N is the set of all nodes in G which are hidden nodes for a forthcoming outside world (as soon as we are going to nest graphs).

(4) KE(G) := KE ⊆ KN × $\mathcal{EL}$ × KN is the set of all known edges in G.

(5) E(G) := E ⊆ (KN × $\mathcal{EL}$ × N) ∪ (N × $\mathcal{EL}$ × KN) is the set of edges which belong to G; either the source or the target of such an edge has to belong to G, too.

(6) HE(G) := HE ⊆ E is the set of all hidden edges in G.

Furthermore, we will use the following abbreviations for node and edge sets:

(7) CN(G) := KN \ N is the set of all context nodes for G, which are known in G but do not belong to G.

(8) VN(G) := N \ HN is the set of all visible nodes of G.

(9) CE(G) := KE \ E is the set of all context edges of G, which are known in G but do not belong to G.

(10) VE(G) := E \ HE ⊆ ((KN \ HN) × $\mathcal{EL}$ × (KN \ HN)) is the set of all visible edges in G; sources or targets of these edges may not be hidden nodes.

Instantiating the definition of encapsulated graphs with atomic nodes, we get the definition of the set of **flat encapsulated graphs**, i.e. $\mathcal{G}(\mathcal{A})$. ❑

Figure 3 shows the relationships between different kinds of nodes of two graphs G and G' in the form of overlapping circles. Both graphs have sets of nodes which really belong to them (VN ∪ HN = N). Additionally, they may have knowledge about nodes of other graphs within the same context, a forthcoming hierarchical graph. The graph G knows a subset of all of those nodes of G', which are potentially visible for him. "*Potentially visible*" means that G may or may not know all visible nodes of G' (and vice versa). The following figure 4 displays for instance a situation, where two graphs A and B know different subsets of the set of all visible nodes of another graph C.

As we will see later on, visible nodes of G and G’ will become subsets of the nodes of a “surrounding” hierarchical graph H. Furthermore, all context nodes of G and G’ have to be known nodes in the hierarchical graph H (cf. def. 3.5). The same situation holds for edges, which have to fulfill additional constraints concerning their source and target nodes.

The definition of such a graph is incomplete as long as the relationships between a graph and its surrounding context within a larger graph are still undefined. This part of our hierarchical graph model will be revealed in definition 3.5 of complex nodes. Disregarding this kind of incompleteness, points (1) through (4) and (6) through (9) above are rather straight-

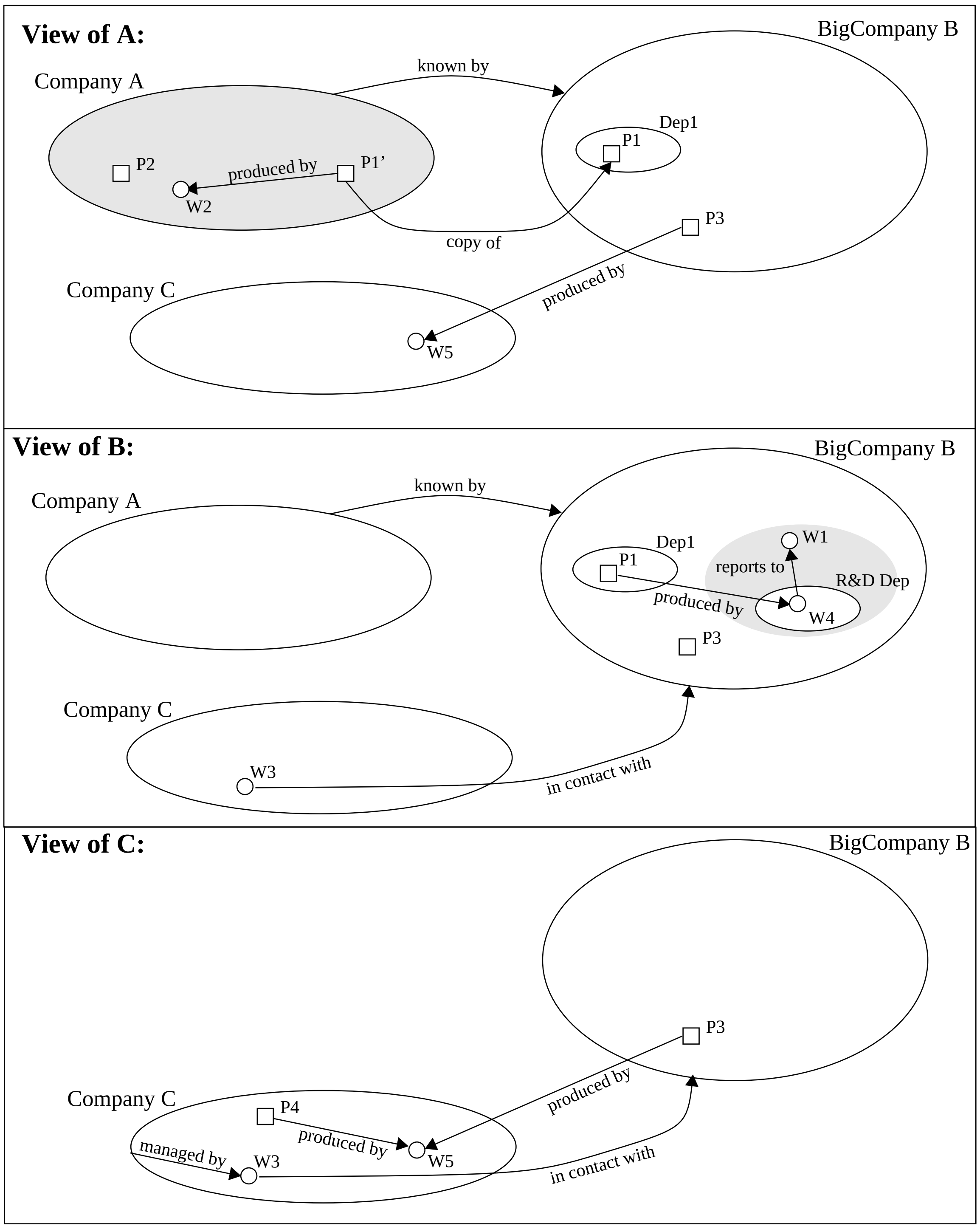

Fig. 4: Different views of hierarchical graphs for different complex nodes.

forward. The requirements in point (5) and (10) for edges need more explanation. They are even to a certain extent a matter of taste. They state that.

- an ordinary edge belongs to an ordinary graph if and only if its source or its target belongs to the graph, too,
- an edge is invisible for the outside world as long as its source or its target are hidden nodes of the graph, and

- even an edge with source or target outside G belongs to the invisible part of G if its target or source is a hidden node in G.

Visible graph boundary crossing edges are the most interesting elements of encapsulated graphs. They do not always have a uniquely defined owner: the graphs of their source and target nodes as well as their smallest common surrounding graph parent may be their owners. This has serious consequences for the definition of graph transformations on hierarchical graphs (in a forthcoming second part of this paper). Having graph elements without uniquely defined owners makes it difficult to maintain the principle of data abstraction that any graph object should be responsible for its own internal state only. Figure 4 gives an example of the "view of the world" for different (sub-)graphs of a hierarchical graph. It is consistent with the following formal definitions of (Big-)Company graphs:

Example 3.4 Definition of (Big-)Company Graphs

The graphs A, B, and C of figure 4 are defined as follows (cf. also figure 1 and example 3.6 which contains the definitions for complex nodes Dep1 and R&D_Dep1 in B):

(1) HN(A) := { P2, W2, P1' },
N(A) := HN(A),
KN(A) := N(A) ∪ { A, C, W5, B, Dep1, P1, P3 },
HE(A) := { (P1', produced_by, W2), (P1', copy_of, P1) },
E(A) := HE(A),
KE(A) := E(A) ∪ { (A, known_by, B), (P3, produced_by, W5) }.

(2) HN(B) := { W1, R&D_Dep1, W4 },
N(B) := HN(B) ∪ { Dep1, P1, P3 },
KN(B) := N(B) ∪ { B, A, C, W3 },
HE(B) := { (P1, produced_by, W4), (W4, reports_to, W1) },
E(B) := HE(B) ,
KE(B) := E(B) ∪ { (A, known_by, B), (W3, in_contact_with, B) }.

(3) HN(C) := { },
N(C) := { W3, P4, W5 },
KN(C) := N(C) ∪ { C, B, P3 },
HE(C) := { },
E(C) := { (C, managed_by, W3), (P4, produced_by, W5), (P3, produced_by, W5) },
KE(C) := E(C) ∪ { (W3, in_contact_with, B) }. ❏

The translation of the picture of figure 4 into the set definitions above is rather straightforward except the treatment of graph boundary crossing edges:

- The known_by edge between graphs A and B may not belong to graphs A and B themselves, since both its source and target do not belong to A or B. But both graphs know themselves and each other and are, therefore, aware of the existence of the known_by edge between them (cf. definition of surrounding hierarchical graph in example 3.6).

- The in_contact_with edge between node W3 in C and B may not belong to graph B (for the same reasons as the known_by edge neither belongs to A nor to B) but it may or may not belong to graph C (its source W3 belongs to C).
- For similar reasons, the produced_by edge between P3 in B and W5 in C might belong to B or to C or to their common surrounding graph (or to all of them). But the definition of graph B says that B has no knowledge about the potentially visible node W5 in C. Therefore, B may neither know nor own the produced_by edge. On the other hand, graph C knows P3 in B and is, therefore, a legal owner of this edge.
- Finally, the copy_of edge between P1' in A and P1 in B has to belong to A as a hidden edge; its source P1' is a hidden node in A which is neither visible for B nor for the surrounding hierarchical graph.

Using the definition of encapsulated graphs with support for information hiding, we are now able to introduce hierarchical graphs, which support *aggregation* of subgraphs.

Def. 3.5 Hierarchical Graphs and Complex Nodes

A triple c := (nid, G, nl) is a **complex node**, in signs $c \in \mathcal{C}$, which contains a **hierarchical graph** G as its internal state, in signs $G \in \mathcal{G}(\mathcal{C})$, if and only if it belongs to the smallest set of elements which fulfill the following conditions:

(1) *nid*(c) := nid $\in \mathcal{NID}$ is a unique node identifier.

(2) *nl*(c) := nl $\in \mathcal{NL}$ is the node's label.

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

(4) G(c) := G $\in \mathcal{G}(\mathcal{C})$ is the node's internal state, a hierarchical graph, which may eventually be a flat (or empty) encapsulated graph.

(5) $\forall\, n \in N(G(c)): CN(G(n)) \subseteq KN(G(c)) \wedge CE(G(n)) \subseteq KE(G(c))$, i.e. all context elements of nodes in c are known in c.

(6) $\forall\, n \in N(G(c)): VN(G(n)) \subseteq N(G(c)) \wedge VE(G(n)) \subseteq E(G(c))$, i.e. all visible elements of nodes which belong to c are known in c and belong to c, too.

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

(8) $\forall\, n \in N(G(c)): KE(G(c)) \cap (KN(G(n)) \times \mathcal{EL} \times KN(G((n))) \subseteq KE(G(n))$, 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 and complex nodes is essentially *recursive*. A complex node contains a hierarchical graph which in turn contains or knows complex nodes. The only restriction, we will get later on, is that "*contains*" relationships between complex nodes and their subnodes form a finite tree (cf. def. 3.7.) Starting with atomic nodes as the most primitive complex nodes guarantees that the defined set $\mathcal{C}$ is not empty.

The definition's requirements (1) through (4) are rather straightforward and should be self-explanatory. The additional restrictions (5) through (8) of definition 3.5 need some explanations. They guarantee that *knowledge* about context elements and especially about visible graph boundary crossing edges is *propagated* to all involved partners (owners) up and down the hierarchy of nested graphs:

- Constraint (5) requires for instance that graph H of example 3.6 knows or even owns all context nodes A, B, C, … of graph A in example 3.4.
- Constraint (6) requires for instance that all visible nodes of B, i.e. Dep1, P1, and P3, belong to H, too.
- Constraint (7) requires for instance that the above mentioned nodes Dep1, P1, and P3 are hidden nodes in H, because B itself is a hidden node of H.
- And constraint (8) requires that A knows the produced_by edge of H from P3 to W5, and that both B and C know the in_contact_with edge between them.

Furthermore, our constraints guarantee that a complex node makes almost no distinctions between its direct subnodes and the visible subnodes of its subnodes. In this way, a complex node gets the allowance to create edges between its direct or indirect visible subnodes as required. Finally, we have to emphasize that the definition above does not exclude the case, where a complex node knows itself, such that edges may connect child nodes with their own parent nodes in a graph hierarchy (e.g. managed_by edge in figure 1 in C).

Example 3.6 Hierarchical Graphs

The hierarchical graph H of figure 1 is defined as follows, such that its internal structure is hidden from the view of any forthcoming outside world:

(1) HN(H) := { A, B, Dep1, P1, P3, C, W3, P4, W5 },
KN(H) := N(H) := HN(H),
HE(H) := { (C, managed_by, W3), (P4, produced_by, W5),
(P3, produced_by, W5), (A, known_by, B), (W3, in_contact_with, B) },
KE(H) := E(H) := HE(H).

Its complex nodes A, B, and C are already defined in example 3.4. But we have still to provide the promised definitions of two complex nodes within B:

(2) HN(Dep1) := { },
N(Dep1) := { P1 },
KN(Dep1) := N(Dep1) $\cup$ { Dep1, R&D_Dep1, W4},
E(Dep1) := HE(Dep1) := { },
KE(Dep1) := { (P1, produced_by, W4) }.

(3) HN(R&D_Dep1) := { },
N(R&D_Dep1) := { W4},
KN(R&D_Dep1) := N(R&D_Dep1) $\cup$ { R&D_Dep1, Dep1, W1 },

HE(R&D_Dep1) := { },
E(R&D_Dep1) := { (W4, reports_to, W1) },
KE(R&D_Dep1) := E(R&D_Dep1). ❏

Based on the definition of hierarchical graphs and complex nodes, we are now able to study conceptual relationships between nodes and graphs. The first one simply states that knowing nodes may be interpreted as "*using*" them. The second one restricts our hierarchical graph data model in such a way that ownership relationships between nodes form a forest, i.e. a set of finite trees. The root of each tree has to be a complex node with empty context.

Def. 3.7 Uses and Contains Relationships

Let $c \in \mathcal{C}$ be a complex node and $x \in \mathcal{C} \cup \mathcal{C} \times \mathcal{EL} \times \mathcal{C}$ be a node or edge:

(1) c **uses** x $:\Leftrightarrow x \in CE(G(c)) \vee x \in CN(G(c)) \setminus \{ c \}$,
i.e. x is a known context element in c unequal to c itself[1].

(2) c **contains** x $:\Leftrightarrow x \in E(G(c)) \vee (x \in N(G(c)) \setminus \{ c \} \wedge \neg \exists n \in N(G(c)): x \in N(G(n)))$,
i.e. x has to be an edge in c or a direct subnode of c unequal to c itself[2].

(3) $\textbf{contains}^+$ ($\textbf{contains}^*$) is the transitive (reflexive) closure of "*contains*".

These relationships between a complex node and other nodes and edges are restricted as follows:

(4) $\forall c \in \mathcal{C}: \{ n \in \mathcal{C} \mid c \textit{ contains}^+ n \}$ is a finite set.

(5) $\forall n, c, c' \in \mathcal{C}: (c \textit{ contains } n \wedge c' \textit{ contains } n) \Rightarrow (c = c')$,
i.e any node has at most one uniquely defined "least upper" owner node.

(6) $\neg \exists c \in \mathcal{C}: c \textit{ contains}^+ c$, i.e. nodes may not contain themselves.

(7) $\forall c \in \mathcal{C}: (\neg \exists c' \in \mathcal{C}: c' \textit{ contains } c) \Rightarrow (CN(G(c)) = \emptyset \wedge CE(G(c)) = \emptyset)$,
i.e. root nodes of hierarchical graphs have an always empty context.

The definition that a graph G **uses** or **contains** nodes and edges may be obtained by replacing any occurrence of "G(c)" by "G" in (1) to (3) above and by deleting "\ { c }" occurrences. ❏

The definition above states that a complex node either uses or contains its known nodes and edges. Two complex nodes may be aware of each other (cyclic use relationships as in example 3.8 are permitted), but they may not contain each other. The restriction to acyclic "*contains*" relationships and the accompanying finiteness requirement makes not only sense from a practical point of view, but it is also needed within the proof of proposition 3.11.

1. A complex node may or may not be an element of its own context node set. In the first case, it is possible to create edges between the node and its children, in the second case not.
2. We have to allow that complex nodes and especially complex node types contain themselves. Otherwise, recursive type definitions wouldn't be possible in subsection 3.3. Furthermore, the condition takes into account that the set N(G(c)) contains also visible nodes of subnodes of c; they have to be excluded for a proper definition of "contains" (cf. def. 3.5, requirement (6)).

Example 3.8 Use and Contains Relationships between Departments, Companies, …

Within this example and all following examples we will assume that the hierarchical graph definitions of example 3.4 and 3.6 are extended to complex node definitions by merely adding missing node identifier and label fields. Then

(1) H *contains* A, B, C, (A, known_by, B), (W3, in_contact_with, B).

(2) B *contains* Dep1, R&D_Dep1, W1, (P1, produced_by, W4).

(3) B *uses* A, C, W3, (A, known_by, B), (W3, in_contact_with, B).

(4) Dep1 *contains* P1.

(5) Dep1 *uses* R&D_Dep1, W4, (P1, produced_by, W4.

(6) R&D_Dep1 *contains* W4, (W4, reports_to, W1).

(7) R&D_Dep1 *uses* Dep1, W1. ❑

3.2 Refinement of Complex Nodes and Hierarchical Graphs

This subsection studies two conceptual relationships between hierarchical graphs which are related to the usual *subgraph* and *graph isomorphism* definitions of other graph data models. The first one, *refinement*, is a very general concept, which comprises the usual subgraph relationship between flat or hierarchical graphs.As a consequence, the concept of isomorphic or better *equivalent* hierarchical graphs is based on refinement: two graphs or complex nodes are equivalent if they refine each other. Both the refinement as well as the equivalence relationship will be introduced under the simplifying assumption that labels of nodes may not be changed. Later on, in subsection 3.3, we will interpret node labels as type identifiers and allow that a node refines another node if its node type is a refinement of the other node's type.

Def. 3.9 Refinement and Equivalence of Atomic Nodes

We assume the existence of a **refinement relationship** *refines* as well as of an **equivalence relationship** $\cong$ with the following properties:

(1) $\forall$ n, n' $\in \mathcal{A}$: (n' *refines* n) $\Rightarrow$ (*nl*(n') = *nl*(n)),
i.e. refinement preserves node labels.

(2) $\forall$ n $\in \mathcal{A}$: n *refines* n,
i.e. every node refines itself.

(3) $\forall$ n, n', n" $\in \mathcal{A}$: (n *refines* n' $\wedge$ n' *refines* n") $\Rightarrow$ n *refines* n",
i.e. the refinement relationship is transitive.

(4) $\forall$ n, n' $\in \mathcal{A}$: (n' $\cong$ n) $\Leftrightarrow$ (n' *refines* n $\wedge$ n *refines* n'),
i.e. the fact that two nodes are equivalent implies that they refine each other and vice versa. ❑

Def. 3.10 Refinement of Complex Nodes and Hierarchical Graphs

Let c, c' $\in \mathcal{C}$ be two complex nodes such that KN(G(c)) and KN(G(c')) are nonempty sets of nodes. The node c' **refines** c, iff:

(1) $nl(c') = nl(c)$,
i.e. the label of a refined node is preserved.

(2) $\exists$ 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'.

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

(4) $\forall n \in N(G(c)) : (f(n) \textit{ refines } n \wedge f(n) \in N(G(c'))) \vee (n = c \wedge f(c) = c')$,
i.e. nodes within c may be replaced by refined versions of themselves within c' and recursion has to be preserved, i.e.

$c \in N(G(c))$ implies $f(c) = c' \in N(G(c')$.

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

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

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

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

Omitting requirement (1) above and replacing all occurrences of G(c) and G(c') by G and G' we obtain a related definition of refinement between hierarchical graphs G and G'. ❑

The definition of refinement of hierarchical graphs (complex nodes) is rather complicated. It includes the usual definitions of monomorphic subgraphs as a special case. It is more liberal such that

- descendent nodes need not be preserved but may be replaced by refinements in turn,
- and the visibility “attribute” of a subnode or an edge may be changed from hidden to visible.

Furthermore, the sets of known, visible, and hidden own nodes and edges may be extended arbitrarily, as a refining graph may contain more nodes and edges than the refined graph.

Figure 5 shows a refinement example. It displays a node C in its upper part and a possible refinement C' in its lower part. The latter one has an additional (atomic) subnode W4, a refined complex subnode (Dep1' with an additional node W3 replaces Dep1), more edges, and it makes its previously hidden node W1 visible. C1' has also an extended knowledge about the inner details of its neighbor B with respect to the complex node Dep2. The Dep2 node is a visible node of B which is not in the known context of C but exists nevertheless as a potentially visible node. It is, therefore, depicted as a dashed node in the upper part of figure 5 and as a solid node in its lower part.

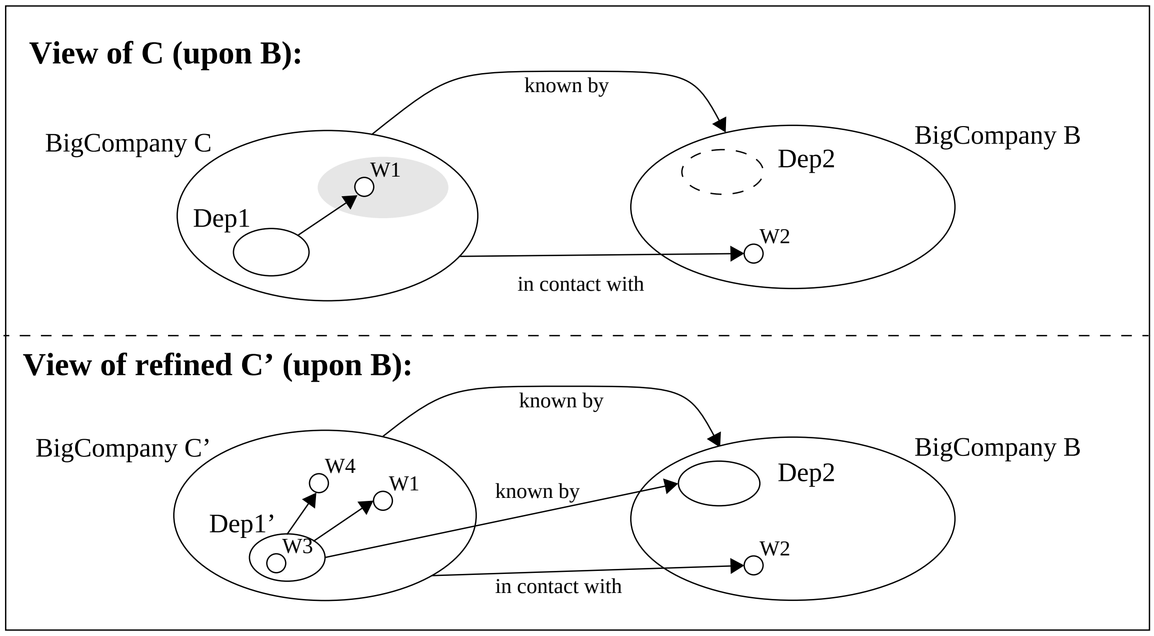

Fig. 5: Complex nodes and their refinements.

Proposition 3.11 Soundness of Refinement Relationship

The refinement relationship between complex nodes is reflexive as well as transitive, i.e.

$\forall$ c $\in \mathcal{C}$: c *refines* c $\wedge$ $\forall$ c, c’, c” $\in \mathcal{C}$: (c *refines* c’ $\wedge$ c’ *refines* c”) $\Rightarrow$ c *refines* c” .

Furthermore, refinement for hierarchical graphs is reflexive and transitive, too.[3]

Proof. The first part of the proof, which shows that every complex node refines itself, i.e.

$\forall$ c $\in \mathcal{C}$: c *refines* c ,

is rather obvious. We simply have to use the identity function

f: KN(G(c)) $\rightarrow$ KN(G(c)) with f(n) = n

in order to view c as a refinement of itself. In that case, all requirements of def. 3.10 are trivially fulfilled.

The second part of the proof shows that

$\forall$ c, c’, c” $\in \mathcal{C}$: (c *refines* c’ $\wedge$ c’ *refines* c”) $\Rightarrow$ c *refines* c”

It requires *induction* over the (finite) *tree depth* of complex nodes with respect to contains relationships. Atomic nodes have a tree depth 0, complex nodes with flat graphs as their internal state a tree depth 1, and so forth. We know already that refinement is transitive for atomic nodes (cf. def. 3.9).Therefore, we can assume that refinement is transitive for all nodes with tree depth *smaller equal* than i-1. Based on this assumption we can prove that refinement is transitive for all nodes with tree depth *smaller equal* i. Thereby, we are able to construct a refinement function f from c to c” by concatenating given refinement functions from c to c’ and from c to c” such that the constraints of definition 3.10 are fulfilled:

3. Both properties depend on each other, since hierarchical graphs contain complex nodes , and complex nodes consist of hierarchical graphs.

(1) $nl(c) = nl(c') = nl(c'')$.

(2) With f_1: KN(G(c)) $\rightarrow$ KN(G(c')), f_2: KN(G(c')) $\rightarrow$ KN(G(c")) being the refinement functions from c to c' and from c' to c" we will use

$f := f_1 \circ f_2$ with $(f_1 \circ f_2)(x) := f_2(f_1(x))$

as the refinement function from c to c".

(3) $\forall n \in CN(G(c)): f_1(n) \in CN(G(c')) \wedge f_2(f_1(n)) \in CN(G(c''))$
$\Rightarrow f(n) = f_2(f_1(n)) = n \in CN(G(c''))$.

(4) $\forall n \in N(G(c))$: n, $f_1(n)$, $f_2(f_1(n))$ have a tree depth smaller equal i-1. Therefore,

n *refines* $f_1(n) \wedge f_1(n)$ *refines* $f_2(f_1(n)) \Rightarrow$ n *refines* $f(n) = f_2(f_1(n)) \in N(G(c''))$
$\vee$ $(c = n) \Rightarrow (f_1(n) = c') \Rightarrow (f_2(f_1(n)) = c'')$.

(5) $n \in VN(G(c)) \Rightarrow f_1(n) \in VN(G(c')) \Rightarrow f(n) = f_2(f_1(n)) \in VN(G(c''))$.

(6) $(n_1, e, n_2) \in CE(G(c)) \Rightarrow (f_1(n_1), e, f_1(n_2)) \in CE(G(c'))$
$\Rightarrow (f(n_1), e, f(n_2)) = (f_2(f_1(n_1)), e, f_2(f_1(n_2))) \in CE(G(c''))$.

(7) $(n_1, e, n_2) \in E(G(c)) \Rightarrow (f_1(n_1), e, f_1(n_2)) \in E(G(c'))$
$\Rightarrow (f(n_1), e, f(n_2)) = (f_2(f_1(n_1)), e, f_2(f_1(n_2))) \in E(G(c''))$.

(8) $(n_1, e, n_2) \in VE(G(c)) \Rightarrow (f_1(n_1), e, f_1(n_2)) \in VE(G(c'))$
$\Rightarrow (f(n_1), e, f(n_2)) = (f_2(f_1(n_1)), e, f_2(f_1(n_2))) \in VE(G(c''))$.

The proof that refinement is reflexive and transitive for hierarchical graphs follows directly from the proof for complex nodes. ❏

We have shown above that refinement is reflexive and transitive and that it is very similar to the well-known concept of monomorphisms between "flat" graphs. Therefore, refinement should be used to define *equivalence classes* of hierarchical graphs which play about the same role as isomorphism classes for "flat" graph data models[4].

Def. 3.12 Equivalence of Complex Nodes and Hierarchical Graphs

Two **nodes** c, c' $\in \mathcal{C}$ are **equivalent** if they refine each other via the same function f:

$c \cong c' :\Leftrightarrow$ c *refines* c' via a function f $\wedge$ c' *refines* c via f^{-1} .

And two **graphs** G, G' $\in \mathcal{G}(\mathcal{C})$ are **equivalent** if they refine each other:

$G \cong G' :\Leftrightarrow$ G *refines* G' via a function f $\wedge$ G' *refines* G via f^{-1} . ❏

Proposition 3.13 Soundness of Equivalence Relationship

Two equivalent nodes c, c' $\in \mathcal{C}$ differ only with respect to the *actual identifier*s of themselves and their direct and indirect subnodes. Similarly, two equivalent graphs differ only with respect to the identifiers of their nodes.

Proof. We have to use the same kind of induction as in proposition 3.11. Starting with known properties about atomic nodes (cf. def. 3.9) it is easy to show

4. We do not use the terms "graph (mono-)morphism" or "graph isomorphism" over here, since we are not concerned with categories of hierarchical graphs and graph morphisms between them.

- that any subnode n in KN(G(c)) has an equivalent or even identical corresponding subnode n’ in KN(G(c’)) and vice versa, such that n and n’ differ at most with respect to their identifiers (induction about tree depth),
- that corresponding subnodes of c and c’ are either both known context nodes or visible own nodes or hidden own nodes,
- and that any known, visible and hidden own edge of c has a corresponding edge in c’ of the same category (and vice versa). ❑

3.3 Graph Schemata and Schema Consistent Graphs

Within this subsection we will study relationships between graphs and graph schemata as well as between complex nodes and their types. It is convenient to model *graph schemata* themselves as graphs, which have again graphs as their graph schemata. In order to avoid the pitfalls of the “the type Type is an instance of itself” assumption [13], we have to introduce a *layered universe* of nodes and graphs. The lowest layer contains all ordinary nodes and graphs, the next layer all node types and graph schemata, the following layer all types of node types and meta schemata, and so forth. These layers together with a *instance of* relationship across layers a third kind of hierarchical relationships between (complex) nodes and their graphs. In the sequel, we will define the *instance of* relationship precisely and discuss how it is related to the other basic (hierarchical) relationships, aggregation and refinement.

For this purpose, the original alphabets of node identifiers and node labels have to be replaced by a single layered set of node identifiers such that each node has the identifier of its node type as its label.

Def. 3.14 Layered Basic Alphabets

In the sequel, we will need the following symbol sets instead of those of def. 3.1:

(1) $\mathcal{NID} := \bigcup_{(k=0)}^{\infty} ID_k$ is a layered set of **node identifiers**.

(2) $\mathcal{NL} := \mathcal{NID} \setminus ID_0$ is the corresponding layered set of **node types** (labels).

(3) $\mathcal{EL}$ is still an unlayered set of **edge types** (labels). ❑

Def. 3.15 Layered Atomic Nodes and their Types

A tuple n := (nid, tid) is an **atomic node** (type) of a **layer** $\mathcal{A}_k \subset \mathcal{A}$ iff:

(1) *nid*(n) := nid $\in ID_k$ is the node’s unique identifier.

(2) *tid*(n) := tid $\in ID_{k+1}$ is the node’s type,
i.e. the identifier of an atomic node of the next higher layer.

In the sequel, we will use the expression

(3) t = *type*(n) $\in \mathcal{A}_{k+1} :\Leftrightarrow$ *nid*(t) = *tid*(n)

for referencing that node t of the next upper layer which is the type of a given node n. ❑

Using the definition of layered atomic instead of atomic nodes we have to repeat def. 3.5 of hierarchical graphs in order to get a definition of layered hierarchical graphs.

Def. 3.16 Layered Hierarchical Graphs and Complex Nodes

A triple c := (nid, G, tid) is a **complex node** of a **layer** $\mathcal{C}_k \subset \mathcal{C}$ which contains a **hierarchical graph** G of **layer** k as its internal state, in signs $G \in \mathcal{G}(\mathcal{C}_k) \subset \mathcal{G}(\mathcal{C})$, if and only if it belongs to the smallest set of elements which fulfill the following conditions:

(1) *nid*(c) := nid $\in$ ID_k is a unique node identifier of layer k.

(2) *tid*(c) := tid $\in$ ID_{k+1} is the complex node's type of layer k+1, and a function *type* is defined in a similar manner as for atomic nodes: t = *type*(c) $\in \mathcal{C}_{k+1}$:$\Leftrightarrow$ *nid*(t) = *tid*(c).

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

(4) G(c) := G $\in \mathcal{G}(\mathcal{C}_k)$ is the node's internal state, a hierarchical graph of layer k, which may eventually be a flat (or empty) encapsulated graph.

The remaining conditions (5) through (8) of def. 3.5 need not be changed. ❑

Based on the layered universe of hierarchical graphs and (complex) nodes, adapted versions of use, contains, refinement, and equivalence relationships must be introduced by simply replacing any occurrence of $\mathcal{A}$ by $\mathcal{A}_k$, of $\mathcal{C}$ by $\mathcal{C}_k$, and of *nl* (for node label) by *type* (within their old definitions). These adapted definitions and their accompanying propositions are not repeated over here, but will be used from now on to define graph schemata and schema consistent graphs as well as new relationships between them.

Def. 3.17 Graph Schemata and Schema Consistent Graphs

A hierarchical graph S of a layer k+1 serves as a **schema** for another graph G of layer k, i.e. it is a consistent graph with respect to the given schema, iff:

(1) $\forall$ 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).

(2) $\forall$ n $\in$ KN(G) \ HN(G) $\exists$ t $\in$ KN(S) \ HN(S): *type*(n) *refines* t,
i.e. visible (context) nodes may not have invisible type definitions in S.

(3) $\forall$ n : (G *contains* n) $\Rightarrow$ $\exists$ t : (S *contains* t) $\wedge$ (*type*(n) = t) $\wedge$ (G(t) is *schema* of G(n)),
i.e. nodes belonging to G must have types belonging to S.

(4) $\forall$ (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 known declaration in the schema S, such that its actual source and target types are refinements of the given types in S.

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

(6) $\forall$ (n_1, e, n_2): (G *contains* (n_1, e, n_2))
$\Rightarrow$ $\exists$ (t_1, e, t_2): (S *contains* (t_1, e, t_2)) $\wedge$ (*type*(n_1) *refines* t_1) $\wedge$ (*type*(n_2) *refines* t_2),
i.e. any edge in G must have a corresponding declaration in the schema S. ❑

The definition above simply requires that a graph contains (knows) only nodes and edges which have own (known) type definitions in its schema. Furthermore, we exclude the case, where a visible graph element has a hidden definition in the schema. That is the underlying principle of *information hiding*: you cannot access elements of data structure or operations on data structures as long as you are not aware of the existence of the corresponding definitions. But note that the requirements above do not prohibit that a graph contains hidden elements with visible definitions in the schema. These elements may move from the hidden part of a graph to its visible part (and vice versa).

Conditions (2) and (5) are rather restrictive. They do not permit that a graph contains nodes, whose type definitions are not part of its schema but imported into the schema. This has the consequence that Company, BigCompany, Department, and R&D_Dep have all their own Worker and Product type definitions. As already discussed within subsection 2.2, these type definitions may be equivalent to each other (except the fact that they are nodes within complex nodes of different types) or they may be refinements of each other. The following example defines the graph schemata for our running example, omitting all needed definitions of atomic node types and refines as well as equivalence relationships between them.

Example 3.18 Definition of Graph Schemata

The graphs of example 3.4 and 3.6 are schema consistent with the following definitions of schema graphs with Root being the schema definition of the overall graph H of example 3.6. Furthermore, R&D_Dep.Researcher is assumed to be a refinement of Department.Worker as well as R&DDep.Prototype is assumed to be a refinement of Department.Prototype:

```
(1)  HN(Company) :=      { },
     VN(Company) :=      {  Company.Worker, Company.Product },
     CN(Company) :=      {  Company },
     HE(Company) :=      {  (Company.Product, copy_of, Company.Product) },
     VE(Company) :=      {  (Company.Product, produced_by, Company.Worker),
                            (Company, managed_by, Company.Worker),
                            (Company.Worker, in_contact_with, Company) },
     CE(Company) :=      {  (Company, known_by, Company) }.

(2)  HN(Department) :=   { },
     VN(Department) :=   {  Department.Worker, Department.Product },
     CN(Department) :=   {  Department, Company, Company.Worker },
     HE(Department) :=   {  (Department.Product, copy_of, Department.Product) },
     VE(Department) :=   {  (Department.Product, produced_by, Department.Worker),
                            (Department, managed_by, Department.Worker),
                            (Department.Worker, in_contact_with, Department),
                            (Department.Worker, reports_to, Company.Worker) },
     CE(Department) :=   {  (Company, known_by, Company),
                            (Company, managed_by, Company.Worker),
                            (Company.Worker, in_contact_with, Company) }.
```

(3) The definition of BigCompany.Department is equivalent to the definition of Department (and, therefore, not repeated over here), except the fact that all occurrences of Department above must be replaced by BigCompany.Department.

(4) HN(R&D_Dep) := { },
VN(R&D_Dep) := { R&D_Dep.Researcher, R&D_Dep.Prototype },
CN(R&D_Dep) := { R&D_Dep, Company, Company.Worker },
HE(R&D_Dep) := { (R&D_Dep.Prototype, copy_of, R&D_Dep.Prototype) },
VE(R&D_Dep) := { (R&D_Dep.Prototype, produced_by, R&D_Dep.Researcher),
(R&D_Dep, managed_by, R&D_Dep.Researcher),
(R&D_Dep.Researcher, in_contact_with, R&D_Dep),
(R&D_Dep.Researcher, reports_to, Company.Worker) },
CE(R&D_Dep) := { (Company, known_by, Company),
(Company, managed_by, Company.Worker),
(Company.Worker, in_contact_with, Company) }.

(5) HN(BigCompany) := { R&D_Dep } ∪ VN(R&D_Dep),
VN(BigCompany) := { BigCompany.Worker, BigCompany.Product,
BigCompany.Department }
∪ VN(BigCompany.Department),
CN(BigCompany) := { BigCompany, Company, Company.Worker, Department}
∪ VN(Department),
HE(BigCompany) := { (BigCompany.Product, copy_of, BigCompany.Product) }
∪ VE(R&D_Dep),
VE(BigCompany) := { (BigCompany.Product, produced_by, BigCompany.Worker),
(BigCompany, managed_by, BigCompany.Worker)
(BigCompany.Worker, in_contact_with, BigCompany) }
∪ VE(BigCompany.Department),
CE(BigCompany) := { (Company, known_by, Company),
(Company, managed_by, Company.Worker),
(Company.Worker, in_contact_with, Company) }
∪ VE(Department).

(6) HN(Root) := { Company, BigCompany, Department }
∪ VN(Company) ∪ VN(Department) ∪ VN(BigCompany),
VN(Root) := { },
CN(Root) := { },
HE(Root) := { (Company, known_by, Company) }
∪ VE(Company) ∪ VE(Department) ∪ VE(BigCompany),
VE(Root) := { },
CE(Root) := { }. ❑

Based on the definition of schema consistent graphs, we are now able to provide the reader with the definition of a *instance_of* relationship between (complex) nodes and their (complex) node types.

Def. 3.19 Instances of Node Types

Let c ∈ $\mathcal{C}_k$ and t ∈ $\mathcal{C}_{k+1}$ be a complex node and a complex node type.

c **instance_of** t :⇔ t = *type*(c) ∧ G(t) is a graph schema for G(c). ❑

The definition above has the consequence that nodes are *proper instances of uniquely defined types* as long as their graph contents is consistent with the type's graph schema contents. Please notice that we do not believe that it is always possible to transform a schema consistent graph into another schema consistent graph with a single rewrite step. Therefore, temporary inconsistencies should be permitted, where *type*(c) = t, but not c *instance_of* t.

Our experiences show that it is rather d*ifficult to develop a proper graph schema* for a class of hierarchical graphs and to check by hand whether a given graph is indeed consistent with its schema definition. The main problems are the subtle interactions between use and refinement relationships and graph boundary crossing edges on the instance as well on the type level. These problems are studied within the following example:

Example 3.20 Instance of Relationships

The hierarchical graph B of example 3.4 (with added type information)

```
HN(B) :=  {  W1:BigCompany.Worker, R&D_Dep1:R&D_Dep,
             W4:R&D_Dep.Researcher },
VN(B) :=  {  Dep1:BigCompany.Department, P1:BigCompany.Department.Product,
             P3:BigCompany.Product },
CN(B) :=  {  B:BigCompany, A:Company, C:Company, W3:Company.Worker },
HE(B) :=  { (P1, produced_by, W4), (W4, reports_to, W1) },
VE(B) :=  { },
CE(B) :=  {  (A, known_by, B), (W3, in_contact_with, B) }.
```

is consistent with its schema definition in example 3.18

```
HN(BigCompany) :=  { R&D_Dep } ∪ VN(R&D_Dep),
VN(BigCompany) :=   {  BigCompany.Worker, BigCompany.Product,
                       BigCompany.Department }
                   ∪  VN(BigCompany.Department),
CN(BigCompany) :=  {  BigCompany, Company, Company.Worker, Department }
                   ∪  VN(Department),
HE(BigCompany) :=  {  (BigCompany.Product, copy_of, BigCompany.Product) }
                   ∪  VE(R&D_Dep),
VE(BigCompany) :=  {  (BigCompany.Product, produced_by, BigCompany.Worker),
                      (BigCompany, managed_by, BigCompany.Worker)
                      (BigCompany.Worker, in_contact_with, BigCompany) }
                   ∪  VE(BigCompany.Department),
CE(BigCompany) :=  {  (Company, known_by, Company),
                      (Company, managed_by, Company.Worker),
                      (Company.Worker, in_contact_with, Company) }
                   ∪  VE(Department).
```

Considering the 6 constraints of def. 3.17 we have to check that:

(1) All nodes of HN(B) have type definitions in HN(BigCompany) ∪ VN(BigCompany):
BigCompany.Worker ∈ VN(BigCompany),
R&D_Dep ∈ HN(BigCompany),
R&D_Dep.Researcher ∈ VN(R&D_Dep) ⊂ HN(BigCompany).

(2) All nodes of VN(B) have type definitions in VN(BigCompany):
BigCompany.Department, BigCompany.Worker ∈ VN(BigCompany),
BigCompany.Department.Product ∈ VN(Department) ⊂ VN(BigCompany).

(3) All nodes of CN(B) have types which are refinements of types in VN(BigCompany) ∪ CN(BigCompany):
BigCompany, Company, Company.Worker ∈ CN(BigCompany).

(4) All edges of HE(B) have type definitions in HE(BigCompany) ∪ VE(BigCompany) such that the types of their source/target nodes are refinements of the source/target type in their type definitions:
(W4:R&D_Dep.Researcher, reports_to, W1:BigCompany.Worker) is compatible with (R&D_Dep.Researcher, reports_to, Company.Worker) ∈ VE(R&D_Dep) and VN(R&D_Dep) ⊂ HE(BigCompany) under the additional (reasonable) assumption that BigCompany.Worker *refines* Company.Worker,
(P1:BigCompany.Department.Product, produced_by, W4:R&D_Dep.Researcher) is matched by (BigCompany.Product, produced_by, BigCompany.Worker) ∈ VE(BigCompany) under the assumption that BigCompany.Department.Product *refines* BigCompany.Product and that R&D_Dep.Researcher *refines* BigCompany.Worker[5].

(5) All edges of VE(B) have compatible edge type definitions in VE(BigCompany):
VE(B) is the empty set.

(6) All edges of CE(B) have compatible edge type definitions in VE(BigCompany) ∪ CE(BigCompany):
(A:Company, known_by, B:BigCompany) is compatible with the edge type definition (Company, known_by, Company) ∈ CE(BigCompany), due to the fact that we know that BigCompany *refines* Company, and
(W3:Company.Worker, in_contact_with, B:BigCompany) is compatible with the type definition (Company.Worker, in_contact_with, Company) ∈ CE(BigCompany), but not with (BigCompany.Worker, in_contact_with, BigCompany) ∈ VE(BigCompany).

To summarize, the main problem for "type checking" hierarchical graphs are graph boundary crossing edges. Their "foreign" sources/targets must have types, which are refinements of known types. This requires either very elaborate refinement relationships between (atomic) node types or the import (usage) of additionally needed types from other type definitions (by extending the known context of the regarded complex type definition) as well as additional edge type definitions with imported types as source/target types. ❏

5. Otherwise, an additional edge type definition would be necessary with compatible source and target node types.

A still neglected problem with the definitions above is that (complex) nodes of each layer reference (complex) nodes of the next upper layer as their types. As a consequence, an *infinite number of layers* must be defined before nodes of the bottom layer may be constructed. In order to circumvent this problem, only a finite number of layers must be defined by users of the graph data model. All remaining layers get a *default definition* as follows:

Def. 3.21 Default Completion of Layers

All those layers of our universe of complex nodes, types, meta types etc. which are of no practical relevance have the following definition: $\mathcal{C}_k := \{ t_k \}$ with a single (meta) type t_k:

(1) $nid(t_k) := id_k \in ID_k$ is an appropriately chosen identifier.

(2) $tid(t_k) := id_{k+1} = nid(t_{k+1})$, the identifier of the default (meta) type of the next layer.

(3) $G(t_k) := (\{ t_k \}, \{ t_k \}, \emptyset, E_k, E_k, \emptyset)$ with $E_k := \{ t_k \} \times \mathcal{EL} \times \{ t_k \}$. ❏

Proposition 3.22 Soundness of Default Layer Construction

The default types t_k, t_{k+1}, … are indeed proper complex nodes with respect to definition 3.16.

Proof. The first two requirements of definition 3.16 are guaranteed by steps (1) and (2) of definition 3.21. The third requirement deals with atomic nodes and their embedding into the set of complex nodes and is, therefore, not important for the proof. Requirement (4) of definition 3.16 is fulfilled, since $G(t_k) \in \mathcal{G}(\{t_k\}) = \mathcal{G}(\mathcal{C}_k)$. The following requirement (5) (from def. 3.5) is true, since t_k is the only node in $N(G(t_k))$ and $CN(G(t_k)) = \emptyset$, $CE(G(t_k)) = \emptyset$. In a similar way, all remaining conditions (6) through (8) (from def. 3.5) may be checked using the fact that $VN(G(t_k)) = N(G(t_k))$, $VE(G(t_k)) = E(G(t_k))$, $HN(G(t_k)) = \emptyset$, and $HE(G(t_k)) = \emptyset$.

Finally, the "minimal size" requirement for $\mathcal{C}_k$ is also fulfilled: $\mathcal{C}_k$ has to be at least a singleton set due to the fact that any node has a type on the next higher layer. Finally, it is easy to check that all conditions of definition 3.7 concerning contains relationships are valid, too. ❏

The proposition above neglects the question whether the definition of these default layers allows the construction of arbitrary hierarchical graphs on the highest user defined layer. Therefore, we have to show

- that any node t_k is consistent with its type definition t_{k+1} of the next higher layer,
- and that any constructible complex node c of the highest user defined layer is consistent with the single type definition of the lowest default layer.

Proposition 3.23 Usefulness of Default Layer Construction

Let $\mathcal{C}_k = \{ t_k \}$ be the lowest default layer of definition 3.21. Then

(1) $\forall\, c \in \mathcal{C}_{k-1} : c$ *instance_of* t_k .

(2) $\forall\, i \geq k : t_i$ *instance_of* t_{i+1} .

Proof. We have to show for any constructible complex node c of layer k-1 that G(c) is a schema consistent graph with respect to $G(t_k)$. Furthermore, we have to show that $G(t_i)$ is schema consistent with $G(t_{i+1})$ for any $i \geq k$.

In order to prove the proposition's first part, the six conditions of definition 3.17 must be checked:

(1) $\forall$ n $\in$ KN(G(c)): *type*(n) = $t_k \in$ KN(G(t_k)), with t_k being the only type on layer k.

(2) $\forall$n $\in$ KN(G(c)) \ HN(G(c)): *type*(n) = $t_k \in$ KN(G(t_k)) \ HN(G(t_k)).

(3) $\forall$ n $\in$ N(G(c)): *type*(n) = $t_k \in$ N(G(t_k)).

(4) $\forall$ (n_1, e, n_2) $\in$ KE(G(c)):
(*type*(n_1), e, *type*(n_2)) = (t_k, e, t_k) $\in$ { t_k } $\times \mathcal{EL} \times$ { t_k } = KE(G(t_k)).

(5) $\forall$ (n_1, e, n_2) $\in$ KE(G(c)) \ HE(G(c)):
(*type*(n_1), e, *type*(n_2)) = (t_k, e, t_k) $\in$ { t_k } $\times \mathcal{EL} \times$ { t_k } = KE(G(t_k)) \ HE(G(t_k)).

(6) $\forall$ (n_1, e, n_2) $\in$ E(G(c)):
(*type*(n_1), e, *type*(n_2)) = (t_k, e, t_k) $\in$ { t_k } $\times \mathcal{EL} \times$ { t_k } = E(G(t_k)).

The proof of the proposition's second part is analogous to the proof of its first part. Any occurrence of "c" above must be replaced by "t_i" and any occurrence of "t_k" by "t_{i+1}". ❑

The proposition above shows that the construction of default layers is consistent with the definition of graph schemata and especially that the lowest default layer does not impose any restrictions onto the construction of hierarchical graphs and complex nodes on the first user defined layer.

4. Useful Extensions and Related Work

Up to now, the underlying data model of flat graphs was a very primitive one. Thereby, we were able to keep definitions manageable and focus our interest on the new hierarchy and layering concepts. It is the purpose of this subsection to discuss a number of useful and straightforward *extensions* on an informal level, especially concerning more complex forms of edges.

But before discussing the more complex subject of extended edge definitions, we would like to suggest one possibility how to incorporate *node attributes* into our data model. It is based on the following definition of attribute values, types, and meta types:

- V_0 := set of all possible attribute values,
- V_1 := set of attribute types with $V_1 = type(V_0)$,
- V_k := set of meta attribute types of layer k with $V_K = type(V_{k-1})$.

Furthermore, the definition 3.16 of encapsulated graphs of layer k has to be extended such that edges may not only have known nodes as their targets but also attribute values or types of layer k. In that way, ordinary edges from nodes to values of V_0 are nothing else but node attributes, edges from node types to values of V_1 are attribute declarations, and so forth.

The resulting data model has some similarities with the underlying data model of Hyperlog [18][6] and it is very similar to that one of the knowledge representation language Telos [14]. In Telos, both nodes and attribute values are modelled as so-called "tokens", and binary

6. Hyperlog models graph schemata as graphs, too, but as far as we know, it does not support meta schemata or graph boundary crossing edges.

relationships between tokens play the role of edges and attributes 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. An equivalent extension of our graph data model has about the following form:

- The definition 3.1 (see also def. 3.14) of edge labels must be replaced by a layered set of edge identifiers as follows: $EID := \bigcup_{(k=0)}^{\infty} EID_k$.
- The modeling of edges as ternary relations must be replaced by modeling them as quaternary relations of the following form:

  (edge-id, source-node-id, edge-type-id, target-node-id).

An edge (id, s, tid, t) on layer k is then a legal instance of an edge (tid, st, mid, tt) on layer k+1, iff: *type*(s) = st $\wedge$ *type(t) = tt* (cf. def. 3.17 of graph schemata).

Such an extended hierarchical data model distinguishes even *parallel edges* of the same type between two given nodes by means of their unique identifiers. Having introduced edge identifiers, the question arises whether edges should be first-class objects like nodes such that they are subjects for aggregation, refinement, and attribution. Up to now, we believe that edges should be kept as simple as possible. Otherwise, the distinction 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 [12] and EDGE [16]. There exists even the graph data model of Hy+ [3], which makes almost no distinction between ordinary edges and contains relationships. 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 many graph data models in literature, with some of them being more than 15 years old [20]. Comparing them with the data model presented here, the main differences are that

- almost all of them — except Telos [14] and Hyperlog [18] — do not support a uniform treatment of graphs and graph schemata,
- many of them prohibit graph boundary crossing edges or do not reason about properties of graph boundary crossing edges in detail,
- and all but “pin graph” like models [9] neglect the importance of distinguishing between internal hidden nodes and externally visible nodes which may be referenced by the outside world.

Furthermore, it has to be noticed that object-oriented data models do have a lot of similarities with our graph data model. They support aggregation, inheritance, associations (binary or n-ary relationships) as well as data abstraction. Their major weaknesses are that they do not support meta modeling and that they come without any formal definitions. Approaches like OMT [21], Fusion [2], ADM3 [6] or Objectory [10] do not offer precise answers to the following questions, which were studied over here:

- How do inheritance and aggregation interact with each other in the case, where a refined aggregate type contains more specific entries than its supertype (from a rigid type theoretic point of view this often occurring situation violates the principles of subtyping and cannot be modeled)?
- How do visibility graphs (related to our use relationship between complex nodes) interact with aggregation and inheritance?
- What about associations between subobjects which belong to different objects; to which extent are they allowed and do not violate the principle of information hiding?
- And what about the semantics of so-called modules or subsystems and their relationships to aggregation?

Furthermore, all above mentioned approaches — except ADM3 [6] — disregard the principle of information hiding on the level of complex objects and subsystems. And even ADM3 with its support for subsystems with well-defined interfaces and hidden internal details does not study the interactions between information hiding, complex object boundary crossing associations, and inheritance.

Nevertheless, we have to admit that *our data model is incomplete* in comparison to object-oriented data models as long as integrity constraints, queries, and especially update operations on encapsulated hierarchical graphs are not taken into account. Update operations on hierarchical graphs should be defined as graph rewrite rules, which preserve all inherently existing integrity constraints of hierarchical graphs as well as all application specific schema consistency constraints. It is future work to develop such a new kind of graph rewrite rules, define their intended semantics as binary relations over hierarchical graphs (as suggested in [4] and [11]) and to extend the principle of refinement/subtyping to graph rewrite rules.

This means that we have to develop a new concept of *graph object types* which are complex node types together with a set of 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 subtyping presented over here is extended to behavioral subtyping.

It is our hope that the new data model of hierarchical graphs together with a forthcoming refinement concept for graph rewrite rules may be combined with a recently developed concept of *graph transformations on distributed graphs*, where different graphs share common subgraphs via so-called interface graphs [24]. A first attempt of incorporating the idea of information hiding into the framework of distributed graph transformations is subject of ongoing research activities [24].

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] Coleman D., Arnold P. et al.: *Object-Oriented Development: The Fusion Method*, Prentice Hall International Editions, Prentice Hall (1994)

[3] 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

[4] Ehrig H., Engels G.: *Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems*, Technical Report 93-34, Dept. of Computer Science, Leiden University (1993)

[5] 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

[6] Firesmith D.G.: *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach*, John Wiley & Sons (1993)

[7] Göttler H., Himmelreich B.: *Modelling of Transactions in Object-Oriented Databases by Two-level Graph Grammars*, in: Abstract Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, 151-156

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

[9] 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

[10] Jacobson I.: *Object-Oriented Software Engineering: A Use Case Driven Approach*, revised fourth printing, Addison-Wesley (1994)

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

[12] 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

[13] Meyer A.R., Reinhold M.B.: *'Type' is Not a Type: Preliminary Report*, in: Proc. 13th ACM Symp. POPL '86, 287-295

[14] 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

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

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

[17] Parisi-Presicce F., Piersanti G.: *Multilevel Graph Grammars*, in: Proc. 20th Int. Workship on Graph-Theoretic Concepts in Computer Science, WG '94, LNCS 903, Springer Verlag (1995), 51-64

[18] 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

[19] Pratt T.W.: *Pair Grammars, Graph Languages and String-to-Graph Translations*, in: Journal of Computer and System Sciences, vol. 5, Academic Press (1971),560-595

[20] 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

[21] Rumbaugh J., Blaha M. et al.: *Object-Oriented Modeling and Design*, Prentice Hall (1991)

[22] Schneider H.J.: *On Categorical Graph Grammars Integrating Structural Transformations and Operations on Labels*, in: Theoretical Computer Science (TCS) 109, Elsevier Science Publ. B.V. (1993), 257-274

[23] Schürr A.: *Specification of Graph Translators with Triple Graph Grammars*, in: Tinhofer G. (ed.): Proc. WG 94, Int. Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 903, Springer Verlag (1995), 151-163

[24] Schürr A., Taentzer G.: *DIEGO, another Step towards a Module Concept for Graph Transformations*, in: Montanari U. et al. (ed.): Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier Science Publ. (1995)

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