Definition of an Encapsulated Hierarchical Graph Data Model Static Aspects, Part 1

G. Busatto G. Engels
Leiden University, Dept. of Computer Science
Niels Bohrweg I
P.O. Box 9512 2300 RA Leiden, The Netherlands
{busatto,engels}@wi.leidenuniv.nl

Abstract

Graph grammars have been successfully used as a formalism for the specification of realistic problems but, as far as specification-in-the-large activities are concerned, they still have some deficiencies. In particular, most of them only support the use of flat graphs, whereas for certain applications hierarchical graphs would be a more suitable modelling tool. Furthermore, there is still the need for a graph grammar module concept that allows to split large specifications into smaller sub-specifications. We want to address this problem by applying object-oriented concepts to develop a hierarchical graph data model that supports a suitable module concept for graph grammars.

In this paper, we present the first step in the definition of our encapsulated hierarchical graph (EHG) data model. We introduce the notion of EHG that supports complex nodes (i.e. nodes with an encapsulated graph as their content), edges, encapsulated graphs (i.e. graphs that support importing and exporting of nodes and edges), hierarchical structuring of complex nodes (through a node-subnode relationship), and appropriate conditions for exporting elements of a node along a hierarchy of nodes.

We illustrate the introduced notions by modelling the underlying data structure of a WWW application.

An advantage of graph grammar approaches over existing object-oriented approaches is that the first allow a better coupling between the definition of data constraints and operations. Therefore our data model should also give some insight on how to achieve such integration between constraints and operations in object-oriented data models.

Contents

1 Introduction

In this paper we present the first step in the definition of an encapsulated hierarchical graph (EHG) data model. One of the inspiring ideas for this work is that of integrating concepts from the graph grammar world and from the software engineering and object-oriented world into a single data model. The resulting data model would not only have the advantage of incorporating useful features from different approaches to data modelling, but it should also provide some new ideas on how to deal with some of their drawbacks. In the sequel we explain which concepts we have taken into account and how we plan to develop the full EHG data model.

1.1 Software Engineering and Programming Languages

In order to manage the increasing size and complexity of software systems, almost all programming methodologies encourage the decomposition of large programs into smaller pieces, called *modules*, each one implementing a specific functionality. For example, a compiler could contain a lexical analyzer module, a parser module, a symbol table management module, and so on.

The evolution of programming languages has seen the progressive introduction of language constructs to support modularization in software development. Pascal's procedures and functions, for example, are a construct that allows to group together instructions with a common purpose.

A more advanced kind of module construct can be found, for example, in languages like Modula2 (see for example [Wir 85]) or Ada (see [Dod 83]). A Modula2 module may contain variable declarations, data type declarations, procedures and some initialization code. Only the elements that are explicitly exported are visible to outer modules while the non-exported elements are hidden. An `import` specification at the beginning of a module specifies which entities exported by external modules are to be imported (i.e. used) by the current module.

Summarizing, the main ideas of modularization in software engineering are the following:

- Every module should implement a well defined functionality inside a system.
- Every module has an *interface* that specifies which of the entities that it defines are to be *exported* to other modules and which externally defined entities are to be *imported* from other modules for internal use. The exported entities (for instance data type or procedure declarations) represent the *protocol* to be used by an external agent to use access the module's functionality.
- Any entity defined inside a module that is not exported is hidden to the outside world and is only for internal use of the module itself (this is the principle of *encapsulation* or *information hiding*). Hidden entities (for example procedure bodies) are used by the module to implement its functionality.

We will use these ideas in our definition of the EHG data model.

1.2 Object-Orientation

Object-orientation (see for example [KA 90]) is a software development technology that has become particularly popular in the 1990s.

We focus our attention on OMT (see [Rum 91]), one of the best-known approaches to object-oriented modelling. OMT is an object-oriented software development method that encompasses all the steps from the definition of a conceptual model of a software system to its implementation. A model consists actually of three sub-models: the object model (describing the static properties of a system), the dynamic model (describing the possible states and state transitions of the system), the functional model (describing the data transformations performed by a system).

The basic idea of object-orientation is to model a real world situation or a software system as a set of entities called *objects*. Objects have an internal state, usually described by means of *attributes*. Objects can interact with each other by exchanging *messages*. Upon receiving a message, an object selects and executes one of its *methods*. A method performs some computation and produces some output value and/or modifies its owner object's internal state.

For example, in the implementation of a window graphical user interface, there could be an object associated to each window. Such an object could have attributes including, for instance, the window's position (x and y coordinates on the screen) and the window's dimensions (its width and height), it could have methods like `Move` (to move the window to a new position on the screen) or `Size` (to modify the window's dimensions).

We give a summary of some concepts of object oriented modelling in OMT that we are interested to incorporate in our EHG data model.

- **Objects.** An *object* is an abstraction representing an entity of the real world or a software component. An object is characterized by an internal state (described by its *attributes*) and its possible behaviour (described by its *methods*).

- **Links.** A *link* is some relation between objects. Links can be binary, ternary or higher order, depending on the number of objects involved in the link. Links can also be considered as tuples of objects.

- **Classes.** An (*object*) *class* is a collection of objects with the same attributes and methods. A class can also be thought of as a pattern from which it is possible to generate objects with the same internal structure.

- **Associations.** An *association* is a collection of links with the same structure, i.e. whose involved objects orderly belong to the same classes.

- **Aggregation.** *Aggregation* is a particular type of association, allowing the definition of aggregate object classes, i.e. of classes whose objects can[1] contain sub-objects from other classes.

- **Inheritance.** *Inheritance* is a relation between classes that allows to define more specific (refined) classes from more general ones. If C_1 and C_2 are two classes and C_2 refines C_1 then we say that C_1 is a *superclass* of C_2 and that C_2 is a *subclass* of C_1. Class C_2 *inherits* all the attributes and methods of class C_1. Furthermore class C_2 can add new attributes or methods to the ones it inherits from C_1. C_2 can also override attributes and/or methods that have been inherited from C_1 by redefining them. Inheritance is therefore a mechanism that allows defining new classes from more general ones by adding new features to them.

[1]or must, according to possible constraints

The named concepts of OMT object modeling will play an important role in the definition of our EHG data model as we will illustrate in 1.4.

While OMT, like other object-oriented methods, claims that there is a close coupling between the definition of data constraints and operations, this is in fact restricted to the syntactic coupling of operation names to data constraints within class specifications. What is missing is a semantic coupling, in the sense that the definition of the functionality of operations (in the functional model) obey the definitions of data constraints (in the object model). Some insight on how to address this problem should be provided by the world of graph-grammar based specification.

1.3 Graph-Grammars

Graph-grammars and graph transformation systems have been used by computer scientists for about 25 years in many theoretical and practical application fields. There are experiences with the use of graph-grammar based specification languages in modelling realistic problems, for example [And 96, Zam 96] using the PROGRES graph programming language ([NS 91, Sch 91a, Sch 91b, Zün 92, Zün 95]). In [EMRS 96] (pp. 37–38), an application of graph-grammars in an educational software engineering game is presented. For more examples of graph-grammars applications the reader can refer to [CER 79, ENR 83, ENRR 87, EKR 91, CEER 96].

One interesting feature of the graph-grammar approach to data structures specification is that they allow to integrate the definition of static and dynamic aspects. For example, in a PROGRES specification we can define graph schemata (i.e. node and edge types, cardinality constraints on edges) and then write rewrite rules that transform a schema consistent graph into another schema consistent graph (i.e. constraint preserving graph transformations).

Unfortunately, graph transformation systems still have some deficiencies (cf. [NS 96]):

1. Many of them only support the use of flat graphs, whereas for certain applications hierarchical graphs would be a more suitable modelling tool.
2. Even in graph transformation systems that support the use of hierarchical graphs it is often not possible:
    - To hide some elements of a subgraph, thus allowing some kind of data encapsulation.
    - To define edges between subnodes of a given node (we call such edges *subgraph crossing edges*), although several object oriented data models (like OMT) support such edges (as links between sub-objects of a given object).
3. There is still a need for a suitable module concept for graph-grammars, in order to split large specifications into smaller sub-specifications.

As far as point 1 above is concerned, some authors have already considered the possibility of defining graph rewrite systems on hierarchical graphs. For example, in [Pra 79] a kind of hierarchical graph (called *H-graph*) and graph-grammars (*H-graph grammars*) on them are presented. More examples can be found in [HLW 92, PP 95]. Although the data models proposed in the cited papers support hierarchical graphs, none of them supports both information hiding and subgraph crossing edges (point 2).

As far as point 3 above is concerned, there are some proposals for a graph-grammar module concept. In [EE 95], Ehrig and Engels introduce an abstract framework to study the possibility of applying some "modularization" ideas to the world of graph-grammars. In [KK 96], Kreowski and Kuske introduce *transformation units*, which allow to build modules of related rules whose semantics are binary relations between graphs. In such a framework, a derivation step can be performed either by applying a rewrite rule or by "calling" an entire transformation unit.

In defining our EHG data model we try to support the abovementioned features by taking inspiration from the software engineering and object orientation world.

1.4 Guidelines for the Definition of the EHG Data Model

We devised the following guidelines for the definition of our encapsulated hierarchical graph (EHG) data model:

1. Nodes in an EHG play a similar role as objects in OMT: Each node has an internal structure that may include:
   - Attributes.
   - An internal graph, hence the name *hierarchical.* Notice that the relationship between a node and the nodes of its internal graph correspond to an aggregation link/association in OMT.
   - Operations to be performed on attributes or on the node's internal graph. These operations should be specified through rewrite rules.
2. Edges represent links/associations (hyper-edges should be supported in the case of ternary or higher order links).
3. Each element inside a node can be *hidden* (private) or *visible* (public, exported). Each node/object can access (import) visible elements of other nodes. On the other hand, according to the afore-mentioned software engineering notion of *encapsulation*, a node is not allowed to access hidden elements of other nodes.
4. Inheritance, a powerful concept of object-orientation, should also be introduced in our model, allowing to *refine* a node's internal structure by adding new attributes and/or operations and by enlarging its internal graph.

From the point of view of graph-grammars, the benefits of such an approach would be the following:

1. The development of a graph-grammar module concept.
2. The study of rewrite rules on hierarchical graphs, which have been scarcely investigated so far.

As far as the object-oriented world is concerned, the new data model should offer a model how to write specifications with a better integration between the description of static and dynamic properties of data.

1.5 Scope and Organization of the Paper

In this paper we do a first step towards the definition of our EHG data model. We deal with the static aspect of EHG's, namely:

- *Nodes* and *edges*.
- *Encapsulated graphs*, as a means to describe a node's internal structure.
- Organization of nodes into a hierarchical structure by means of *structuring trees*.
- Definition of scoping rules for import/export of elements between nodes inside a hierarchy.

In this paper we do not yet consider attributes of nodes and the possibility to introduce an inheritance-like relationship between nodes: their study is postponed to further research. Also operations on EHG's are not supported by the current EHG data model and they will be the topic of future research, too.

This paper is organized as follows: In section 2 we introduce a running example that will serve to illustrate our definitions. In section 3 we give a formal definition of our notion of EHG. In section 4 we compare our data model with existing work and we sketch future development for the EHG model.

2 The Running Example

We describe a simple modelling situation where hierarchical graphs can be used as a modelling tool and we exploit it in the following sections as a running example to introduce our definition of encapsulated hierarchical graphs. We assume that the reader has an intuitive idea of what a hierarchical graph is, i.e. a graph where nodes may contain other nodes and edges. In the next section we will provide a formal definition of hierarchical graphs.

2.1 Hypertexts and the World Wide Web

Our example deals with hypertexts and their use in the World Wide Web (WWW). For the reader's convenience we recall some concepts and terminology here.

The WWW is made of a number of sites, i.e. by individual computer systems storing different types of documents and communicating with each other through the internet. By means of appropriate software applications (so-called *browsers*) a user can see this computer network as a unified source of information and can navigate among the documents stored in the WWW without being aware of its underlying structure.

One important class of documents to be found on the WWW is represented by hypertexts, that are normally stored as texts in HTML. HTML (HyperText Markup Language) is a language that permits to describe the logical structure of a WWW document as well as all navigating information related to it. Such navigating information consists of links from one HTML document to other WWW documents or even to specific points inside WWW documents that allow browsers to access the data stored in the WWW in a nonlinear way. When a browser, running on some computer on the internet, is required to display a document, it requests it from the site that provides it, it receives the HTML text for the document and translates it into a suitable visual representation (different browsers may display the same document differently). An introduction to HTML can be found in [NCSA].

For our example it is sufficient to consider a situation where only HTML coded hypertexts are stored in the WWW (we will mention some other types of resources available on the WWW shortly). Furthermore, the only elements of HTML that are of interest for us are *anchors*, i.e. language constructs that permit to set up links between two documents. An anchor can specify:

- A place inside a document to which a browser can jump (in this case we call the anchor a *target anchor*).
- A link to some object in the WWW (in this case we call the anchor a *hyperlink anchor*).
- Both of the above cases.

A target anchor has a `name` attribute whose value can be used to identify it among the anchors of a specific page. A hyperlink anchor has a `href` attribute that specifies an object on the WWW that must be accessed by a browser if it is required to follow that hyperlink. The `href` attribute contains a so-called URL (Uniform Resource Locator). We consider only a simplified type of URL for our example, namely only URL's of the form

`http://`<*site-address*>[<*path*><*filename*>[`#`<*target-anchor-name*>]]

will be allowed. The prefix `http` indicates the *access scheme* or *protocol* to be used to access the specified object. The only protocol we allow here is `http`, i.e. the HyperText Transfer Protocol, used to access files on a WWW server. Other possible protocols are, for example:

- `file`: access a file on the system running the browser.
- `ftp`: access a file on an anonymous ftp server.
- `news`: access a usenet newsgroup.

The remaining part of the URL specifies a site address, an HTML file on that site and an anchor inside that HTML file. A browser will use such an URL to retrieve the HTML document, format it for output and display it starting at the position specified by the anchor. If no anchor is specified the browser will display the document from its beginning. If the page's filename is not specified either, the browser will retrieve the site's default page (its *homepage*) and go to its beginning.

HTML possesses a rich set of document structuring constructs which we will not consider here, assuming that a document is made of a sequence of pieces of text interspersed with anchors.

2.2 Maintaining Consistency in the Web

Suppose to have a number of WWW sites, each providing a set of HTML documents (pages). If, for example, a document is removed from a site, then all documents containing references to it contain dangling hyperlinks. This is a common problem that arises within the more general issue of *distributed hypertext infostructures maintenance* (see [Fie 94]).

Current solutions to this problem involve the use of *spiders*, i.e. of programs that can traverse the web and check for inconsistencies between documents. In [Fie 94] such a program, called MOMspider, is presented.

We hint at a possible research direction for an alternative solution: An infostructure can be modelled as a hierarchical graph satisfying appropriate constraints and all possible modifications to it should be defined as constraints preserving operations on hierarchical graphs. This idea has to be fully developed and by now will only serve as a means to provide an example of data modelling with EHG.

2.3 Modelling the Web with EHG

We want to build a model of a world of web sites by representing information that enables us to:

1. Know all link-dependences between pages and between sites. For example, if a page is removed, we would like to have at hand the information needed to update all pages that contained a link to it.
2. Store information about mirror sites of one site and use it, for example, to redirect requests to a non responding site to one of its mirrors.

We therefore need a data model to represent this situation and hierarchical graphs seem quite appropriate for our purposes, namely:

- Sites can be represented as nodes. The "is a mirror of" relationships between sites can be represented by edges between site nodes.

- HTML pages can be represented as nodes inside site nodes (therefore site nodes are *complex nodes*, i.e. nodes with an internal graph). Every site node must contain a default page node, its *homepage*.
- The contents of pages can be in turn represented as a sequence of nodes of two kinds: anchor nodes and text nodes (i.e. nodes representing any portion of text between two subsequent anchors).
- The normal reading sequence of a page is represented by edges linking each (anchor or text) node to its successor.
- For each hyperlink anchor node there will be an edge connecting it to its target anchor. If an hyperlink's URL specifies a target page but no target anchor inside it, then an edge between the hyperlink anchor node and the target page node is drawn. If the hyperlink does not even specify a target page, then the hyperlink node is connected with an edge to the target site node.

Figure 1, depicting a graph with three site nodes and seven documents, is meant to give a first idea of our example and of the proposed modelling technique. At this stage we will not give a formal definition of the type of graphs that we are using, assuming that figure 1 is intuitive enough.

For the sake of readability, text nodes are depicted as empty boxes while anchors are depicted as full boxes. We refer to nodes of figure 1 by using a dotted notation. Therefore, `S1.P2` denotes the page node `P2` inside the site node `S1`. As already said, in our graph representation only some structural information about the setting of pages, sites and their mutual relationships is considered. Therefore, most of the information contained in the original HTML documents is not modelled. Appendix A is a supplement to this section, containing more precise information about our running example.

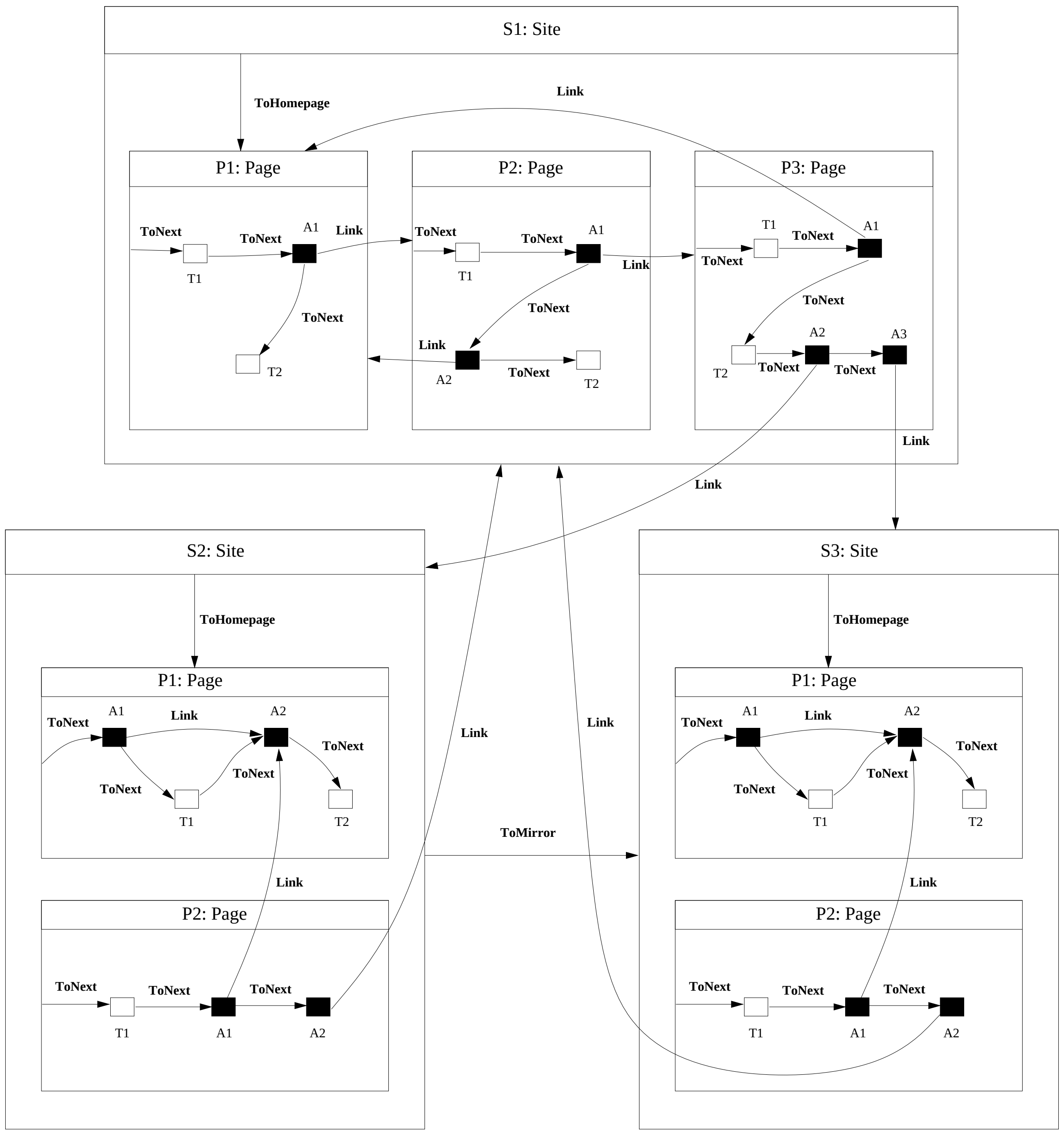

Figure 1: An example containing three site nodes.

3 Encapsulated Hierarchical Graphs

In this section we give a formal definition of encapsulated hierarchical graphs. It is organized as follows:

- In subsection 3.1 we introduce some basic concepts, i.e. the alphabets of *node identifiers*, *node labels* and *edge labels*, *atomic nodes* and *edges*.
- In subsection 3.2 we will introduce *encapsulated graphs* and *complex nodes*.
- In subsection 3.3 we introduce *trees*, which serve to describe the hierarchical structure of an EHG, and *information spreading conditions*, which describe how knowledge about elements of an EHG can be spread along its structure.
- Finally, in 3.4, we will use all the aforementioned concepts to define *encapsulated hierarchical graphs.*

3.1 Basic Concepts

When dealing with encapsulated hierarchical graphs, we will always use three different alphabets to specify:

- Nodes' identity, i.e. every node will have a unique identifier.
- Nodes' and edges' type, i.e. every node and every edge will be labeled with a symbol denoting its type.

Definition 3.1.1 (Basic alphabets) In the sequel we will need three alphabets, namely:

1. $\mathcal{NID}$, the alphabet of *node identifiers*,
2. $\mathcal{NL}$, the alphabet of *node labels*,
3. $\mathcal{EL}$, the alphabet of *edge labels.*

Remark 3.1.2 For our running example we use the following alphabets:

- The set of node identifiers is

  $$\mathcal{NID} := \{\texttt{S1}, \texttt{S1.P1}, \texttt{S1.P2}, \texttt{S1.P3}, \texttt{S1.P1.T1}, \ldots\}$$

  i.e. the set of all strings that are well-formed node names according to the dotted notation informally introduced in 2.3.
- $\mathcal{NL} := \{\texttt{Site}, \texttt{Page}, \texttt{Text}, \texttt{Anchor}\}$. Recall that in figure 1 nodes of type `Text` are depicted as empty rectangles, nodes of type `Anchor` are depicted as full rectangles whereas the types of the other nodes are written explicitly on the nodes' labels. Notice also that we don't distinguish between target and hyperlink anchor in our model.
- $\mathcal{EL} := \{\texttt{ToNext}, \texttt{Link}, \texttt{ToHomepage}, \texttt{ToMirror}\}$. Edge labels are written near edges in figure 1.

We can now define atomic nodes as nodes that "do not contain any subnodes".

Definition 3.1.3 (Atomic nodes) Given a set of node identifiers $\mathcal{NID}$ and set of node labels $\mathcal{NL}$, a set $\mathbf{AN} \subseteq \mathcal{NID} \times \mathcal{NL}$ is a *set of atomic nodes* on $\mathcal{NID}$ and $\mathcal{NL}$ iff

$$\forall n = (i, l), n' = (i', l') \in \mathbf{AN}.(i = i') \Rightarrow (n = n')$$

i.e. the identifier of each node is unique in $\mathbf{AN}$. If $n = (i, l) \in \mathbf{AN}$ (for some set of atomic nodes $\mathbf{AN}$), we write $i = \mathbf{nid}(n)$ and $l = \mathbf{nl}(n)$ and we call n an *atomic node.*

Remark 3.1.4 In our example the set of atomic nodes is

$$\mathbf{AN} := \{(\texttt{S1.P1.T1}, \texttt{Text}), (\texttt{S1.P1.A1}, \texttt{Anchor}), (\texttt{S2.P2.A1}, \texttt{Anchor}), \ldots\}$$

i.e. all nodes of type `Text` and `Anchor` are atomic nodes. It is easy to verify that each atomic node has a unique identifier.

In the sequel, to simplify our notation, we will often use a node's identifier instead of the node itself, i.e. we will write, for example, `S1.P1.T1` instead of (`S1.P1.T1`, `Text`). Although this notation can be imprecise, it does not introduce any inconsistency since nodes are uniquely determined by their identifiers.

If we have a set $\mathbf{N}$ (whose elements' internal structure is not important here) and a set of edge labels $\mathcal{EL}$, we can build *binary, labelled, directed edges* from them.

Definition 3.1.5 (Edges) Given a set $\mathbf{N}$ and a set of edge labels $\mathcal{EL}$, then an *edge* on $\mathbf{N}$ and $\mathcal{EL}$ is a triple (s, l, t), with $s, t \in \mathbf{N}$ and $l \in \mathcal{EL}$, i.e. an edge is any $e \in \mathbf{N} \times \mathcal{EL} \times \mathbf{N}$. Instead of (s, l, t) we will often use the more intuitive notation $s -l\rightarrow t$.

Remark 3.1.6
Two example edges from figure 1 are: $(\texttt{S3.P2.A1}, \texttt{Anchor}) -\texttt{Link}\rightarrow (\texttt{S3.P1.A2}, \texttt{Anchor})$ and $(\texttt{S1.P2.A1}, \texttt{Anchor}) -\texttt{Link}\rightarrow (\texttt{S1.P2.A2}, \texttt{Anchor})$.

3.2 Encapsulated Graphs and Complex Nodes

In our running example we have informally introduced a hierarchical graph to represent some aspects of the underlying structure of the WWW. If such a modeling technique were to be used in a practical application the resulting graph would be very large (due to the size of the real WWW). On the other hand, it is hardly likely that a single application should need a representation of the whole WWW. We think that a reasonable way to address this issue is to use some formal method, which allows us to:

- Define local *views* of the entire graph describing the WWW. For example "the view of site `S1`" (used by the site's manager) or "the view of page `S2.P1`" (used by the page's author).
- Associate to every view specific permissions to query/modify only some elements of the entire graph. For example, in the view of site `S2` it seems reasonable that we can:
  - Query *and* modify page `S2.P1` (`S2` owns it).
  - Query *but not* modify site `S1` (some page inside `S2` has a reference to it).
  - Neither query nor modify node `S1.P3` (it is a page owned by another site and not referenced from `S2`).

- Decide, for each element of such a view, whether other views can query (see) it or not. For example, if we associate a view to page `S2.P1`, such a view should allow other views to see its internal anchor `S2.P1.A2` (it is referenced from page `S2.P2`) while `S2.P1.A1` could be kept for internal use only.

We have decided to associate a partial view of a hierarchical graph to each of its nodes. A node with an internal view will be called a *complex node* and its internal view will be represented by a so-called *encapsulated graph* (EG). Atomic nodes, which have already been defined, can be considered as complex nodes with an empty internal view. We will introduce EG's first.

An EG is made of a set of nodes and a set of labeled directed edges. An element (node or edge) of an EG G can be *local* (owned by G) or *context* (owned by another graph but known and usable for query operations by G). In our example, node `S2.P2.A2` should be a local node of the graph associated to node `S2.P2` while node `S1` should be a context node to enable page `S2.P2` to draw a `Link` edge to it (the operation of creating or removing an edge does not modify the edge's ends, therefore creating edge `S2.P2.A2` $-$`Link`$\rightarrow$ `S1` can be considered a query operation on `S1`).

Furthermore, each element of an EG G can be *visible* (G allows information about that element to be seen by the outside world) or *hidden.* A visible element of one EG can be a context element of some other EG whereas a hidden element cannot (hence the term *encapsulated*). In our example, node `S2.P1.A2` should be a visible node of the graph of `S2.P1` (a target anchor is made public in order to be used by other pages) and a context node of the graph of `S2.P2` which contains a `Link` edge to it.

We now give our formal definition of EG's.

Definition 3.2.1 (Encapsulated graphs) Let $\mathbf{N}$ be a given set (of nodes) and $\mathcal{EL}$ a given set of edge labels, then an *encapsulated graph* over $\mathbf{N}$ and $\mathcal{EL}$ is a tuple

$$G = (\mathbf{N}_G, \mathbf{E}_G, \mathbf{vis}_G, \mathbf{loc}_G)$$

such that:

1. $\mathbf{N}_G \subseteq \mathbf{N}$ is the set of *nodes* of G.

2. $\mathbf{E}_G \subseteq \mathbf{N}_G \times \mathcal{EL} \times \mathbf{N}_G$ is the set of *edges* of G.

3. $\mathbf{vis}_G : \mathbf{N}_G \oplus \mathbf{E}_G \rightarrow \{\mathbf{true}, \mathbf{false}\}$ is a predicate stating whether an element of G is *visible* or *hidden.*

4. $\mathbf{loc}_G : \mathbf{N}_G \oplus \mathbf{E}_G \rightarrow \{\mathbf{true}, \mathbf{false}\}$ is a predicate stating whether an element of G is *local* or *context.*

5. $\forall e = (s, l, t) \in \mathbf{E}_G . \mathbf{loc}_G(e) \Rightarrow (\mathbf{loc}_G(s) \vee \mathbf{loc}_G(t))$, i.e. if e is a local edge of G, then at least one of its ends must be a local node of G.

6. $\forall e = (s, l, t) \in \mathbf{E}_G . \mathbf{vis}_G(e) \Rightarrow (\mathbf{vis}_G(s) \wedge \mathbf{vis}_G(t))$, i.e. if e is a visible edge of G, then both ends of e must be visible nodes of G.

We denote the *empty encapsulated graph* $(\emptyset, \emptyset, \emptyset, \emptyset)$ with $\emptyset_{\mathrm{EG}}$.

Before we proceed we introduce some useful abbreviations. Given an EG $G = (\mathbf{N}_G, \mathbf{E}_G, \mathbf{vis}_G, \mathbf{loc}_G)$, we let:

- $\mathbf{HN}_G := \{n \in \mathbf{N}_G | \mathbf{vis}_G(n) = \mathbf{false}\}$ be the set of *hidden nodes* of G,
- $\mathbf{VN}_G := \{n \in \mathbf{N}_G | \mathbf{vis}_G(n) = \mathbf{true}\}$ be the set of *visible nodes* of G,
- $\mathbf{LN}_G := \{n \in \mathbf{N}_G | \mathbf{loc}_G(n) = \mathbf{true}\}$ be the set of *local nodes* of G,
- $\mathbf{CN}_G := \{n \in \mathbf{N}_G | \mathbf{loc}_G(n) = \mathbf{false}\}$ be the set of *context nodes* of G,
- $\mathbf{HE}_G := \{e \in \mathbf{E}_G | \mathbf{vis}_G(e) = \mathbf{false}\}$ be the set of *hidden edges* of G,
- $\mathbf{VE}_G := \{e \in \mathbf{E}_G | \mathbf{vis}_G(e) = \mathbf{true}\}$ be the set of *visible edges* of G,
- $\mathbf{LE}_G := \{e \in \mathbf{E}_G | \mathbf{loc}_G(e) = \mathbf{true}\}$ be the set of *local edges* of G,
- $\mathbf{CE}_G := \{e \in \mathbf{E}_G | \mathbf{loc}_G(e) = \mathbf{false}\}$ be the set of *context edges* of G.

Furthermore, for ease of notation, we let:

- $\mathbf{HLN}_G := \mathbf{HN}_G \cap \mathbf{LN}_G$,
- $\mathbf{VLN}_G := \mathbf{VN}_G \cap \mathbf{LN}_G$,
- $\mathbf{HLE}_G := \mathbf{HE}_G \cap \mathbf{LE}_G$,
- and so forth.

The whole situation (and the corresponding notation) is summarized in the tables of figure 2.

Nodes		
	HN	**VN**
LN	**HLN**	**VLN**
CN	**HCN**	**VCN**

Edges		
	HE	**VE**
LE	**HLE**	**VLE**
CE	**HCE**	**VCE**

Figure 2: Notation for hidden/visible and local/context property of nodes and edges.

Example 3.2.2 In figure 3 we show the EG associated to page `S2.P1` of our running example. Notice that we have refined our graphical notation to distinguish between local/context and hidden/visible elements. This is done by drawing all elements of a graph inside a rectangular area that is partitioned into four regions corresponding to all the possible combinations of values of hidden/visible and local/context property. The properties of an edge are visually determined by the position of the black bullet associated to it. Let G be the EG of figure 3, then:

- $\mathbf{HLN}_G = \{\texttt{S2.P1.T1}, \texttt{S2.P1.T2}, \texttt{S2.P1.A1}\}$,
- $\mathbf{VLN}_G = \{\texttt{S2.P1.A2}\}$,
- $\mathbf{HCN}_G = \{\texttt{S2.P1}\}$,
- $\mathbf{VCN}_G = \emptyset$,

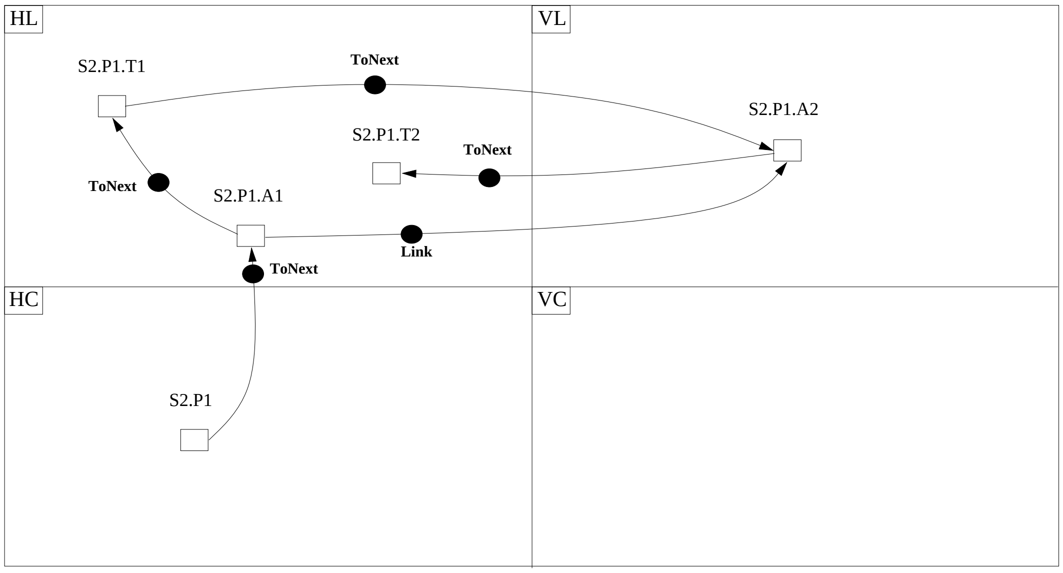

Figure 3: The encapsulated graph associated to page `S2.P1`.

- $\mathbf{HLE}_G = \{$`S2.P1`$\,-$`ToNext`$\rightarrow$`S2.P1.A1`, `S2.P1.A1`$\,-$`ToNext`$\rightarrow$`S2.P1.A2`, `S2.P1.A2`$\,-$`ToNext`$\rightarrow$`S2.P1.T2`, `S2.P1.A1`$\,-$`Link`$\rightarrow$`S2.P1.A2`$\}$,
- $\mathbf{HCE}_G = \mathbf{VE}_G = \emptyset$.

The distinction between local and context elements will be important when defining operations on EG's (these are not yet supported by our data model). More precisely, an operation on an EG should be allowed to *query* and/or *modify* a local element of an EG but only to *query* a context element.

In the graph G of figure 3, all nodes that represent the internal structure of page `S2.P1` (namely `S2.P1.A1`, `S2.P1.T1`, `S2.P1.A2`, `S2.P1.T2`) are local nodes: operations on G should be allowed, for example, to delete these nodes or to modify their internal attributes (node attributes are not yet supported). The edges inside G describe the possible reading sequences of page `S2.P1` and their creation and/or deletion should be performed by operations on G. Therefore all edges of G are local edges. Such operations associated to G could be executed, say, by an application run by the page's author.

Node `S2.P1` is a context node of G. Following our modelling choices, it should be a local node of the graph of `S2` and modifications on it should only be performed by operations of the graph of `S2` (such operations could be executed by some program run by the site's manager). `S2.P1` is a context node of G in order to allow the existence of edge `S2.P1`$\,-$`ToNext`$\rightarrow$`S2.P1.A1`. The creation and deletion of such an edge should be performed by an operation associated to G.

Any context element of an EG must be an element of some other EG (i.e. it must be *imported*). A node or edge of an EG K can (cannot) be a context element of an EG $L \neq K$ if it is visible (hidden). We let all elements of the graph G of figure 3 be hidden, with the exception of node `A2`, which is visible because it is needed by the EG of page `S2.P2`.

Our next step is to define a set of *complex nodes* **N**, i.e. of nodes that possess an internal view, which is described by an EG on **N**. Atomic nodes are embedded in **N** by considering them complex nodes with an empty graph as internal state. Notice that a complex node can contain an EG, which in turn can contain complex nodes, which can contain other EG's, and so on. Such a chain will eventually stop with EG's containing only atomic nodes.

Definition 3.2.3 (Complex nodes) If we have three alphabets $\mathcal{NID}$, $\mathcal{NL}$, $\mathcal{EL}$ and a set of atomic nodes **AN** over $\mathcal{NID}$ and $\mathcal{NL}$, then a *set of complex nodes* **CN** on $\mathcal{NID}$, $\mathcal{NL}$, $\mathcal{EL}$ and **AN** is a set **CN** such that:

- $\forall n \in \mathbf{CN}.\exists i, G, l.n = (i, G, l)$, where $i \in \mathcal{NID}$, G is an EG on **CN** and $\mathcal{EL}$, $l \in \mathcal{NL}$.
- $\forall n = (i, l) \in \mathbf{AN}.\exists n' = (i, \emptyset_{\text{EG}}, l) \in \mathbf{CN}$, i.e. for each atomic node there is a corresponding complex node.
- $\forall n = (i, G, l) \in \mathbf{CN}.\neg\exists n' = (i', G', l') \in \mathbf{CN}.n' \neq n \wedge i = i'$, i.e. all nodes in **CN** have a unique identifier.

If **N** is a set of complex nodes and $c \in \mathbf{CN}$, $c = (i, G, l)$, we denote i as $\mathbf{nid}(c)$, G as $\mathbf{G}(c)$ and l as $\mathbf{nl}(c)$. Furthermore, we call c a *complex node.*

Example 3.2.4 In figure 4 we display the complex node `S2.P2` of our example. Notice that we use a graphical notation similar to that of figure 3: we have only added a top bar containing the node's identifier and label.

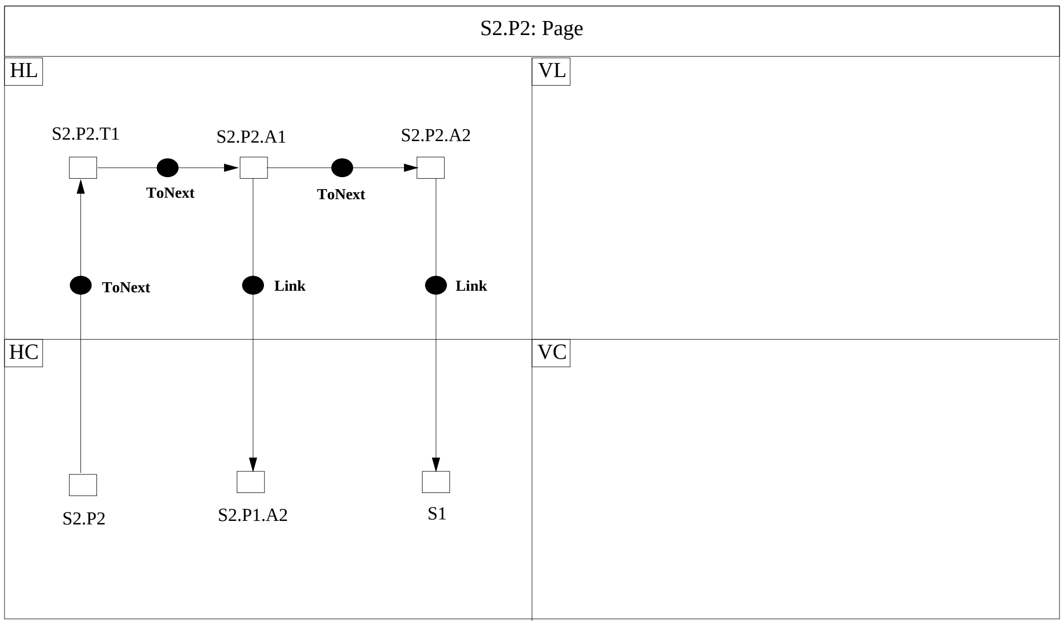

Figure 4: The complex node `S2.P2`.

3.3 Representation of the Hierarchical Structure of Graphs

The final step in our definition of EHG consists of describing how complex nodes can be put together to form an EHG. We need two more tools:

- *trees*, as a means to represent the hierarchical structure of an EHG and
- *information spreading conditions*, i.e. conditions that specify how knowledge about elements of an EHG can be spread through its structure.

Definition 3.3.1 (Trees) A *(directed) tree* is a couple $T = (N, A)$, where N is a set of *nodes* and $A \subseteq N \times N \setminus \{(n, n) | n \in N\}$ is a set of *arcs* such that:

1. $\exists! r \in N. \neg \exists n \in N.(n, r) \in A$, i.e. T has a *root* node that is not the target of any edge of T. We denote the root of T with $\rho(T)$.
2. $\forall n \in N$, if $n \neq \rho(T)$ then there exists a unique path from $\rho(T)$ to n in T, i.e. there exists a unique sequence of nodes $n_1, \ldots, n_k \in N$ $(k > 1)$ such that:
   - for all $i \in \{1, \ldots, k-1\}$, $(n_i, n_{i+1}) \in A$,
   - $n_1 = \rho(T)$, $n_k = n$.

If $n \in N$ and $\neg \exists n'.(n, n') \in A$, then n is a *leaf* of T.

We are going to use trees as structuring graphs for EHG's. An arc (n_1, n_2) in such a tree has the meaning "n_2 is a subnode of n_1", or "n_1 contains n_2". Therefore, in the sequel, if $T = (N, A)$ is a known tree and $n_1, n_2 \in N$, we will often use the more intuitive notation n_1 **contains** n_2 instead of $(n_1, n_2) \in A$.

Example 3.3.2 In figure 5 we display the structuring tree associated to the overall graph of our running example. Notice that we have introduced a fictitious **root** node that contains all other nodes.

We anticipate some intuitive ideas that are useful to understand the following definition 3.3.3. An encapsulated hierarchical graph will consist of a set of complex nodes linked by structuring edges to form a tree. If n is a complex node in such a tree, then:

- $\mathbf{LN}_{\mathbf{G}(n)} := \{n' | n \textbf{ contains } n'\}$, i.e. the local nodes of a node are all and only its subnodes. Local edges of $\mathbf{G}(n)$ only need to satisfy the conditions described in 3.2.1.
- Context elements of $\mathbf{G}(n)$ must be imported from the EG associated to some other node. *Information spreading conditions* (ISC, to be defined shortly) express the constraints that must be satisfied by such imported elements.

There are four ISC's. The first two rules deal with visible context elements while the second two deal with hidden context elements. The difference between these two cases is very slight:

- In all four cases we require that a context element x of a node c be an element of a supernode (parent node) of c.

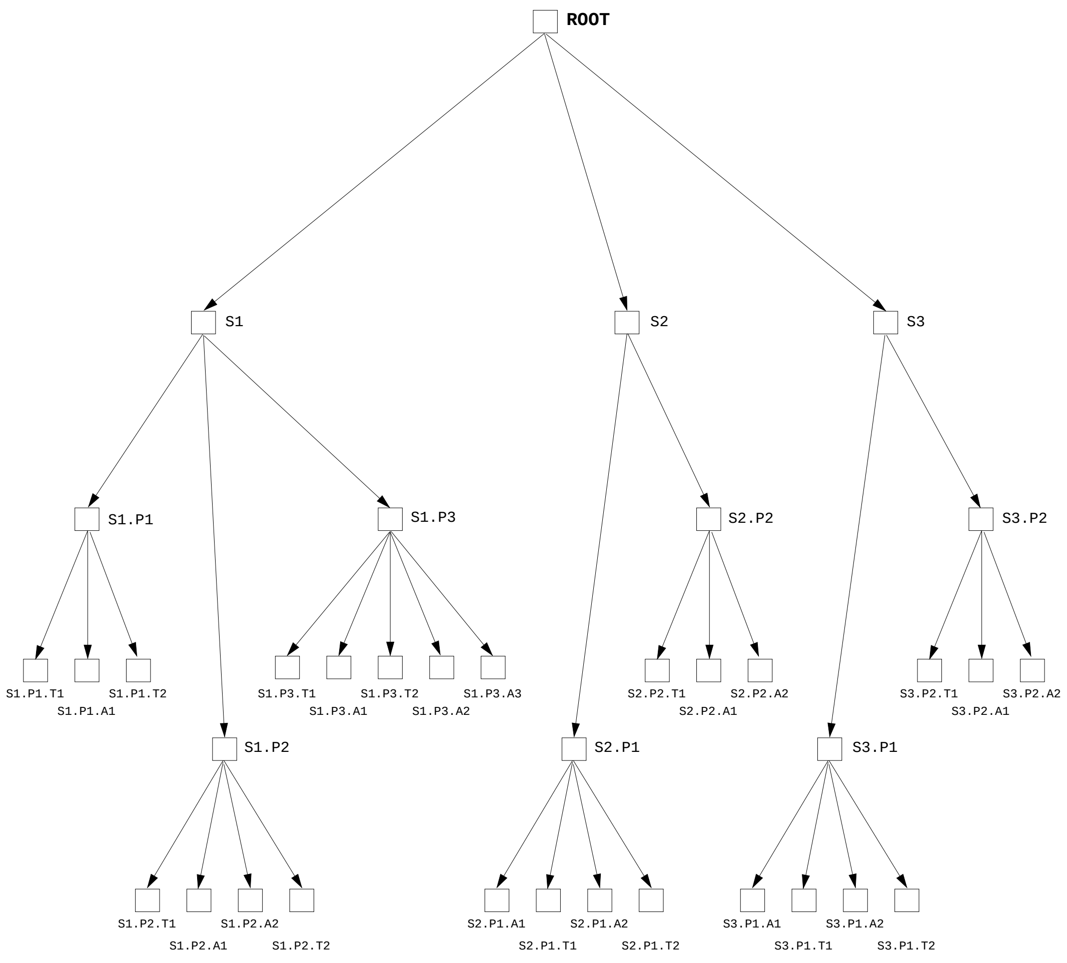

Figure 5: The structuring tree for the WWW example.

- Otherwise we require that x be an element of a subnode (child node) c' of c. In such a case x must a visible element of the subnode (otherwise c would not be allowed to know about x). If we also want x to be a *visible* context element of c, then c' (the subnode providing information about x) must be itself a visible subnode of c (it would not make sense to make visible some information coming from a hidden source).

We now give the formal definition of ISC's.

Definition 3.3.3 (Information spreading conditions) Let $\mathbf{CN}$ be a set of complex nodes and $T = (N, A)$ be a tree such that $N \subseteq \mathbf{CN}$). Then the *information spreading conditions* for T are, for all $c \in N$ and all context elements x of $\mathbf{G}(c)$, the following:

1. $x \in \mathbf{VCN}_{\mathbf{G}(c)} \;\Rightarrow\; (\exists c' \in N.(c' \textbf{ contains } c \wedge x \in \mathbf{N}_{\mathbf{G}(c')}) \vee$
   $(c \textbf{ contains } c' \wedge c' \in \mathbf{VLN}_{\mathbf{G}(c)} \wedge x \in \mathbf{VN}_{\mathbf{G}(c')}))$
   i.e. if x is a visible context node of a node c, then x must either be a node of a parent node of c or a visible node of a visible child node of c.

2. $x \in \mathbf{VCE}_{\mathbf{G}(c)} \;\Rightarrow\; (\exists c' \in N.(c' \;\mathbf{contains}\; c \wedge x \in \mathbf{E}_{\mathbf{G}(c')})\vee (c \;\mathbf{contains}\; c' \wedge c' \in \mathbf{VLN}_{\mathbf{G}(c)} \wedge x \in \mathbf{VE}_{\mathbf{G}(c')}))$
   i.e. if x is a visible context edge of a node c, then x must either be an edge of a parent node of c or a visible edge of a visible child node of c.

3. $x \in \mathbf{HCN}_{\mathbf{G}(c)} \;\Rightarrow\; (\exists c' \in N.(c' \;\mathbf{contains}\; c \wedge x \in \mathbf{N}_{\mathbf{G}(c')})\vee (c \;\mathbf{contains}\; c' \wedge c' \in \mathbf{LN}_{\mathbf{G}(c)} \wedge x \in \mathbf{VN}_{\mathbf{G}(c')}))$
   i.e. if x is a hidden context node of a node c, then x must either be a node of a parent node of c or a visible node of a child node of c.

4. $x \in \mathbf{HCE}_{\mathbf{G}(c)} \;\Rightarrow\; (\exists c' \in N.(c' \;\mathbf{contains}\; c \wedge x \in \mathbf{E}_{\mathbf{G}(c')})\vee (c \;\mathbf{contains}\; c' \wedge c' \in \mathbf{LN}_{\mathbf{G}(c)} \wedge x \in \mathbf{VE}_{\mathbf{G}(c')}))$
   i.e. if x is a visible context edge of a node c, then x must either be an edge of a parent node of c or a visible edge of a child node of c.

Example 3.3.4 We give more intuitive ideas by means of our running example. We suppose that our structuring tree $T = (N, A)$ is the one depicted in figure 5. Let us consider for instance nodes `S2`, `S2.P1`, `S2.P2` and let their internal graphs be the following:

- $\mathbf{G}(\texttt{S2}) = G_1 = (\mathbf{N}_{G_1}, \mathbf{E}_{G_1}, \mathbf{vis}_{G_1}, \mathbf{loc}_{G_1})$, where:
  - $\mathbf{N}_{G_1} = \{\texttt{S1}, \texttt{S2}, \texttt{S3}, \texttt{S2.P1}, \texttt{S2.P2}, \texttt{S2.P1.A2}\}$
  - $\mathbf{E}_{G_1} = \{\texttt{S2} \xrightarrow{\texttt{ToHomePage}} \texttt{S2.P1}, \texttt{S2} \xrightarrow{\texttt{ToMirror}} \texttt{S3}\}$
  - $\forall x \in \{\texttt{S2.P1}, \texttt{S2.P2}, \texttt{S2} \xrightarrow{\texttt{ToHomePage}} \texttt{S2.P1}, \texttt{S2} \xrightarrow{\texttt{ToMirror}} \texttt{S3}\}.\mathbf{loc}_{G_1}(x) = \mathbf{true}$
  - $\forall x \in \{\texttt{S1}, \texttt{S2}, \texttt{S3}, \texttt{S2.P1.A2}\}.\mathbf{loc}_{G_1}(x) = \mathbf{false}$
  - $\forall x \in \mathbf{N}_{G_1} \cup \mathbf{E}_{G_1}.\mathbf{vis}_{G_1}(x) = \mathbf{false}$
- $\mathbf{G}(\texttt{S2.P1}) = G_2 = (\mathbf{N}_{G_2}, \mathbf{E}_{G_2}, \mathbf{vis}_{G_2}, \mathbf{loc}_{G_2})$, where:
  - $\mathbf{N}_{G_2} = \{\texttt{S2.P1}, \texttt{S2.P1.A1}, \texttt{S2.P1.T1}, \texttt{S2.P1.A2}, \texttt{S2.P1.T2}\}$
  - $\mathbf{E}_{G_2} = \{\texttt{S2.P1} \xrightarrow{\texttt{ToNext}} \texttt{S2.P1.A1}, \texttt{S2.P1.A1} \xrightarrow{\texttt{ToNext}} \texttt{S2.P1.T1}, \texttt{S2.P1.T1} \xrightarrow{\texttt{ToNext}} \texttt{S2.P1.A2}, \texttt{S2.P1.A2} \xrightarrow{\texttt{ToNext}} \texttt{S2.P1.T2}\}$
  - $\mathbf{loc}_{G_2}(\texttt{S2.P1}) = \mathbf{false}$
  - $\forall x \in (\mathbf{N}_{G_2} \cup \mathbf{E}_{G_2}) \setminus \{\texttt{S2.P1}\}.\mathbf{loc}_{G_2}(x) = \mathbf{true}$
  - $\mathbf{vis}_{G_2}(\texttt{S2.P1.A2}) = \mathbf{true}$
  - $\forall x \in (\mathbf{N}_{G_2} \cup \mathbf{E}_{G_2}) \setminus \{\texttt{S2.P1.A2}\}.\mathbf{vis}_{G_2}(x) = \mathbf{false}$
- $\mathbf{G}(\texttt{S2.P2}) = G_3 = (\mathbf{N}_{G_3}, \mathbf{E}_{G_3}, \mathbf{vis}_{G_3}, \mathbf{loc}_{G_3})$, where:
  - $\mathbf{N}_{G_3} = \{\texttt{S1}, \texttt{S2.P2}, \texttt{S2.P2.T1}, \texttt{S2.P2.A1}, \texttt{S2.P2.A2}, \texttt{S2.P2.T2}\}$
  - $\mathbf{E}_{G_3} = \{\texttt{S2.P2} \xrightarrow{\texttt{ToNext}} \texttt{S2.P2.T1}, \texttt{S2.P2.T1} \xrightarrow{\texttt{ToNext}} \texttt{S2.P2.A1}, \texttt{S2.P2.A1} \xrightarrow{\texttt{ToNext}} \texttt{S2.P2.A2}, \texttt{S2.P2.A1} \xrightarrow{\texttt{Link}} \texttt{S2.P1.A2}, \texttt{S2.P2.A2} \xrightarrow{\texttt{Link}} \texttt{S1}\}$
  - $\forall x \in \{\texttt{S2.P2.T1}, \texttt{S2.P2.A1}, \texttt{S2.P2.A2}\} \cup \mathbf{E}_{G_3}.\mathbf{loc}_{G_3}(x) = \mathbf{true}$
  - $\forall x \in \{\texttt{S2.P2}, \texttt{S2.P1.A2}, \texttt{S1}\}.\mathbf{loc}_{G_3}(x) = \mathbf{false}$

– $\forall x \in \mathbf{N}_{G_3} \cup \mathbf{E}_{G_3}.\mathbf{vis}_{G_3}(x) = \mathbf{false}$

We can verify that these three nodes respect the ISC's for T. For instance, `S1` $\in \mathbf{HCN}_{\mathbf{G}(\texttt{S2.P2})}$. Then condition 3 of definition 3.3.3 must be satisfied. It actually is because `S1` $\in \mathbf{N}_{\mathbf{G}(S2)}$ and `S2` **contains** `S2.P2`. More intuitively, site `S1` is a hidden context node in page `S2.P1`. It means that either site `S1` is a visible node of a subnode of `S2.P1` (which is not the case) or that `S1` is a node of the graph of a super-node of `S1.P2`. Since `S1` is a context node of site `S2` which contains page `S2.P1`, the required condition is satisfied.

As a second example, we can trace how knowledge about node `S2.P1.A2` is transmitted from page `S2.P1` to page `S2.P2` (see also figure 6). This is done in two steps:

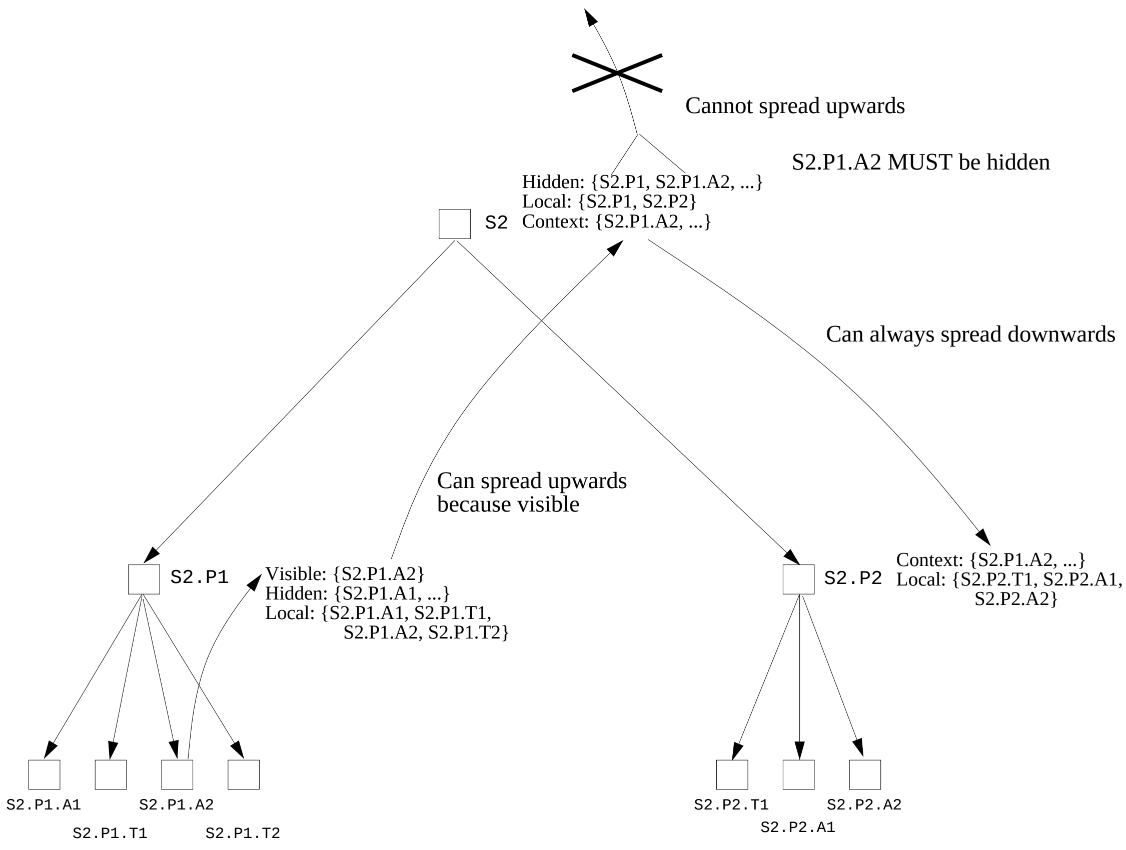

Figure 6: Application of spreading conditions in example 3.3.4.

- First the common parent node `S2` can have `S2.P1.A2` among its context nodes because it is a visible node in `S2.P1`. In other words, `S2.P1.A2` $\in \mathbf{HCN}_{\mathbf{G}(\texttt{S2})}$ means that `S2` and `S2.P1.A2` must satisfy condition 3. It actually does because `S2` **contains** `S2.P1` and `S2.P1.A2` $\in \mathbf{VN}_{\mathbf{G}(\texttt{S2.P1})}$.
- Since `S2.P1.A2` is in $\mathbf{HCN}_{\mathbf{G}(\texttt{S2})} \subseteq \mathbf{N}_{\mathbf{G}(\texttt{S2})}$, both conditions 1 and 3 are satisfied for `S2.P2` and `S2.P1.A2`. Then `S2.P1.A2` can be also in $\mathbf{CN}_{\mathbf{G}(\texttt{S2.P2})}$ and it can be either a visible or hidden node there.

Further verifications are left to the reader.

3.4 Encapsulated Hierarchical Graphs

We are now able to introduce encapsulated hierarchical graphs. Intuitively, an EHG is a tree (i.e. a set of nodes and a **contains** relationship between them satisfying proper conditions) where all nodes have an internal state represented by an encapsulated graph (i.e. they are complex nodes). As has been already partially anticipated, such a tree must satisfy the following conditions:

- The local nodes inside the internal graph of any node c are all and only the sub-nodes of c with respect to T.
- The local edges of the graph of any node c of T only need to satisfy the conditions described in 3.2.1
- The leaves of T must be atomic nodes.
- All ISC's for T (which define constraints on context elements of nodes of T) must be satisfied.

Definition 3.4.1 (Encapsulated hierarchical graphs) Given three alphabets $\mathcal{NID}$, $\mathcal{NL}$, $\mathcal{EL}$ and a set of atomic nodes $\mathbf{AN}$ on $\mathcal{NID}$, $\mathcal{NL}$, an *encapsulated hierarchical graph* on $\mathcal{NID}$, $\mathcal{NL}$, $\mathcal{EL}$ and $\mathbf{AN}$ is a tree $T = (N, A)$ satisfying the following conditions:

1. N is a set of complex nodes on $\mathcal{NID}$, $\mathcal{NL}$, $\mathcal{EL}$ and $\mathbf{AN}$ (therefore every $c \in N$ has an encapsulated graph on N as its internal state).
2. $\forall n \in N$, if n is a leaf, then $n \in \mathbf{AN}$.
3. $\forall n \in N . \mathbf{LN}_{\mathbf{G}(n)} := \{n' \in N | n \ \mathbf{contains} \ n'\}$.
4. All ISC's for T are satisfied.

Fact 3.4.2 *In an EHG $T = (N, A)$, for every $n \in N$, $n \neq \rho(T)$, there exists exactly one node n' such that $n \in \mathbf{LN}_{\mathbf{G}(n')}$.*

Remark 3.4.3 In the proposed EHG data model, we have decided that every node has a view (encapsulated graph) associated to it and that, although nodes can be context elements of more than one view, they must be local elements of exactly one view. In other words, every node in an EHG can be owned by (and, therefore, be modified by the operations associated to) exactly one EG inside that EHG.

A slightly different approach is to be found in [ES 95a] where the visible local nodes of a node n are local nodes of its super-node (if such a super-node exists). The condition that a node can only modify its direct sub-node could prove to be too restricted and could be relaxed following the choices taken in the mentioned paper.

To put this question in terms of our WWW example, we have chosen that, say, no operation associated to node `S2` is allowed to modify anchor `S2.P1.A2` directly. It could achieve this by calling a possible exported operation of page `S2.P1`. If the approach of [ES 95a] had to be chosen, an operation associated to site `S1` could modify (maybe delete?) anchor `S2.P1.A2` inside page `S2.P1` directly.

4 Conclusions and Future Work

In this paper we have introduced first concepts of a new encapsulated hierarchical graph data model. It supports the creation of graphs which consist of a hierarchically structured set of nodes. Nodes have an internal state that is an encapsulated graph. Encapsulated graphs support importing of nodes and edges as well as information hiding.

In an object model in OMT we can define graph-like structures which are comparable to EHG's, in particular:

- The concepts of object/class in OMT correspond to the concept of node in EHG.
- Binary links/associations correspond to edges.
- Aggregation links/associations correspond to the **contains** relation between nodes.

The major differences between the two data models are the following:

- The EHG model does not (yet) support node attributes and operations.
- The EHG model only supports binary edges while OMT allows ternary or higher order links/associations.
- EHG does not allow to define any inheritance-like relation between nodes.
- The EHG data model does not (yet) allow to distinguish between graph instances (objects) and graph schemata (classes).
- OMT only supports encapsulation for attributes and operations inside an object/class. Sub-objects of a given object cannot be hidden and associations can be freely drawn between sub-objects of different objects.

It is the purpose of a forthcoming paper to enhance the EHG model with the following features:

- Node attributes.
- An inheritance-like relation between nodes.
- Support for the definition of graph instances and graph schemata.

A more complex issue is the introduction of rewrite rules on EHG. A powerful feature that they should support is the possibility to modify the internal graph of a node together with the internal graph of some of its subnodes within a single rewrite step.

Thinking of our WWW example, a rewrite rule that removes a page from one site should also update all pages (possibly in other sites) containing references to it. Therefore such a rule should be applied to the overall graph (i.e. it should be a rule of the **root** node) and should be capable to descend into the node hierarchy and update the internal graph of nodes where needed.

The issue of operations on EHG is the subject of ongoing research.

A Supplement to the Running Example

This appendix serves to complete the information about our running example. Some basic knowledge of HTML is assumed here. The few notions that are needed to read the following can be found in the already cited [NCSA].

We give the listing of the HTML documents whose structure is depicted in figure 1. Of course we are not interested in the text parts that they contain. We will therefore denote them with the symbol *text*.

Our example contains three web sites, namely: `S1` (internet address `www.site1.nl`), `S2` (internet address `www.site2.nl`), `S3` (internet address `www.site3.nl`). `S3` is a mirror of `S2`. `S1` contains three WWW pages: `/page1.html`, `/page2.html` and `/page3.html` (referring to nodes `S1.P1`, `S1.P2` and `S1.P3` of figure 1 respectively). Likewise, site `S2` maintains two files: `/page1.html` and `/page2.html` (nodes `S2.P1` and `S2.P2`) and its mirror `S3` two files with the same names (nodes `S3.P1` and `S3.P2`). The (simplified) HTML sources of the seven documents are the following:

`http://www.site1.nl/page1.html:`

```
    text
    <a href="http://www.site1.nl/page2.html"> text </a>
    text
```

`http://www.site1.nl/page2.html:`

```
    text
    <a href="http://www.site1.nl/page3.html"> text </a>
    <a href="http://www.site1.nl/page1.html"> text </a>
    text
```

`http://www.site1.nl/page2.html:`

```
    text
    <a href="http://www.site1.nl/page1.html"> text </a>
    text
    <a href="http://www.site2.nl"> text </a>
    <a href="http://www.site3.nl"> text </a>
```

`http://www.site2.nl/page1.html:`

```
    <a href="http://www.site2.nl/page1.html#A2"> text </a>
    text
    <a name="A2"> text </a>
    text
```

`http://www.site2.nl/page2.html:`

```
    text
    <a href="http://www.site2.nl/page1.html#A2"> text </a>
    <a href="http://www.site1.nl"> text </a>
```

`http://www.site3.nl/page1.html:`

```
<a href="http://www.site3.nl/page1.html#A2"> text </a>
text
<a name="A2"> text </a>
text
```

`http://www.site3.nl/page2.html:`

```
text
<a href="http://www.site3.nl/page1.html#A2"> text </a>
<a href="http://www.site1.nl"> text </a>
```

References

[And 96] M. Andries, *Graph Rewrite Systems and Visual Database Languages*, PhD thesis, Leiden University (NL) 1996.

[CAB 94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, *Object-Oriented Development—The Fusion Method*, Prentice Hall, 1994.

[CER 79] V. Claus, H. Ehrig, G. Rozenberg, *Proc. 1st Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology, International Workshop*, LNCS 73, Springer-Verlag (1979).

[CEER 96] J. Cuny, H. Ehrig, G. Engels, G. Rozenberg (eds.): *Proc. 5th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 1073, Springer Verlag (1996).

[CH 95] A. Corradini, R. Heckel: *A Compositional Approach to Structuring and Refinement of Typed Graph Grammars*, in U. Montanari et al. (ed.): Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier Science Publ. (1995).

[Dod 83] *Reference Manual for the Ada Programming Language*, ANSI/MIL STD 1815A, US DoD (Jan 1983).

[EE 95] H. Ehrig, G. Engels: *Towards a Module Concept for Graph Transformation Systems*, Technical Report 93-34, Dept. of Computer Science, Leiden University (1993).

[EKR 91] H. Ehrig, H.-J. Kreowski, G. Rozenberg (eds.): *Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science*, LNCS 532, Springer Verlag (1991).

[ENR 83] H. Ehrig, M. Nagl, G. Rozenberg (eds.): *Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science, Proceedings*, LNCS 153, Springer-Verlag (1983).

[ENRR 87] H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (eds.): *Proc. 3rd Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 291, Springer-Verlag (1987).

[EMRS 96] H. Ehrig, U. Montanari, G. Rozenberg, H. J. Schneider (eds), *Graph Transformations in Computer Science*, Dagstuhl-Seminar-Report; 155, 09.09.-13.09.96 (9637).

[ES 95a] G. Engels, A. Schürr: *Encapsulated Hierarchical Graphs, Graph Types, and Meta Types.* In A. Corradini, U. Montanari (eds.): Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, August 1995, Volterra (Italy), Electronic Notes in Theoretical Computer Science 1 (1995), 75-84.

[ES 95b] G. Engels, A. Schürr: *Encapsulated Hierarchical Graphs, Graph Types, and Meta Types*, Technical Report 95-21, Department of Computer Science, Leiden University, July 1995.

[Fie 94] R. T. Fielding: *Maintaining Distributed Hypertext Infostructures: Welcome to MOMspider's Web*, in Computer Networks and ISDN Systems (Special Issues) Vol. 27, no. 2, Selected Papers of the First World-Wide Web Conference, Geneva, Switzerland, 25-27 May 1994.

[HLW 92] F. Höfting, T. Lengauer, E. Wanke: *Processing of Hierarchically Defined Graphs and Graph Families*, in Lecture Notes in Computer Science 594, Springer-Verlag (1992).

[KA 90] S. Khoshafian, R. Abnous: *Object Orientation, Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, inc. (1990).

[KK 96] H. J. Kreowski, S. Kuske: *On the Interleaving Semantics of Transformation Units - A Step into GRACE*, in [CEER 96].

[NCSA] *NCSA Beginner's Guide to HTML*:
`http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html`.

[NS 91] M. Nagl, A. Schürr: *A Specification Environment for Graph Grammars*, in [EKR 91], 599-609.

[NS 96] M. Nagl, A. Schürr: *Software Integration Problems and Coupling of Graph Grammar Specifications*, in [CEER 96].

[Pra 79] T. W. Pratt, *Definition of Programming Language Semantics Using Grammars for Hierarchical Graphs*, in [CER 79].

[PP 95] F. Parisi-Presicce, G. Piersanti: *Multilevel Graph Grammars*, in Lecture Notes in Computer Science 903, Springer Verlag (1995).

[Rum 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object Modelling and Design*, Prentice Hall inc. (1991).

[Sch 91a] A. Schürr: *PROGRES: A VHL-Language Based on Graph Grammars*, in [EKR 91], 641-659.

[Sch 91b] A. Schürr: *Operationales Spezifizieren mit Programmierten Graphersetzungssystemen: Formale Definitionen, Anwendungen und Werkzeuge*, Dissertation, RWTH Aachen, Deutscher Universitätsverlag (1991).

[Sch 96] A. Schürr, *Programmed Graph Replacement Systems*, to appear.

[Set 96] R. Sethi, *Programming Languages, Concepts and Constructs* 2$^{\text{nd}}$ edition, Addison Wesley (1996).

[ST 95] A. Schürr, G. Taentzer, *DIEGO, Another Step Towards a Module Concept for Graph Transformation Systems*, in Electronic Notes in Theoretical Computer Science 2 (1995).

[Wir 85] N. Wirth, *Programming in Modula2*, Springer (1985).

[Zam 96] A. Zamperoni, *GRIDS - GRaph-based, Integrated Development of Software: Integrating Different Perspectives of Software Engineering*, in Proceedings of the 18th International Conference on Software Engineering, Berlin (Germany), March 25-29, 1996.

[Zün 92] A. Zündorf: *Implementation of the Imperative/Rule Based Language PROGRES*, TR No. AIB 92-38, RWTH Aachen, Germany (1992).

[Zün 95] A. Zündorf: *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme*, Dissertation, RWTH Aachen (1995).