DHHTGraphs - Modeling Beyond Plain Graphs

Daniel Bildhauer, Jürgen Ebert

University of Koblenz-Landau, Germany
(dbildh,ebert)@uni-koblenz.de

***Abstract*— Graphs are known as a suitable representation of structured data in general and of models in particular. Most common graph databases and storage facilities are based on a relatively simple kind of graphs restricted to binary and often even untyped or non-attributed edges. However, for domain-specific modeling there is also a need for some natural advanced concepts such as n-ary edges, several kinds of hierarchy and distribution of a graph over several sites. While common graph approaches require to simulate those concepts by plain graphs, this paper introduces an enhanced approach offering a seamless integration of these concepts, called Distributed Hierarchical Hyper-TGraphs (DHHTGraphs). Based on a formal definition, DHHTGraphs are described in detail including a metamodeling language and an efficient implementation.**

I. Introduction

In model-driven software development (MDSD), *models* are central artifacts during all development phases and *modeling tools* used to create and store these models. Following the vision of MDSD, modeling languages and modeling tools are primarily influenced by their ability to support generation of code in object-oriented languages. Links are assumed to be implemented by reference attributes, objects are assumed to be atomic, and the models are described as single comprehensive entities. But, regarding the growing importance of models, their expressiveness, maintainability and algorithmic accessibility seems to become more important than the technical, highly automatable step of model implementation in a model-centric development process. Hence, modeling languages should rather aim at a *natural, convenient, expressive and concise representation* of the modeled world than at their suitability for code generation.

In this paper, we claim that a versatile modeling language should allow modeling concepts beyond entities and relations. It should support the natural representation of attributed *n-ary relationships* as well as of *hierarchically structured* elements with several levels of detail. Furthermore, the language should support models that are *distributed* over several sites, enhancing development in cooperative distributed teams.

While *graphs* consisting of vertices and connecting edges are widely used as suitable data structures to represent models, plain graphs provide no natural representation of these advanced concepts. Different kinds of hypergraphs, several forms of hierarchical graphs and some kinds of distributed graphs have been proposed. But, a consistent realization and implementation of distributed hierarchical hypergraphs with typing, attribution and nesting of vertices as well as edges is still missing.

We propose *Distributed Hierarchical Hyper-TGraphs (DHHTGraphs)* as a formal basis for these advanced modeling concepts and develop a compatible repository technology that supports on them. DHHTGraphs allow the partitioning of models into separate parts, which can be handled independently on separate systems but still can be considered as one cohesive model if necessary. Furthermore, hierarchy concepts, namely nesting and levels of detail, are supported for vertices as well as for edges. Finally, the representation of n-ary relationships, called hyperedges in graph literature, with named ends is supported. The use of DHHTGraphs saves the extra work needed to emulate these aspects again and again by simulating these properties using flat graphs. Thus, a higher level of abstraction can be used in modeling which is seamlessly supported by the infrastructure.

In this paper, we sketch the formal definition of DHHTGraphs and present a corresponding metamodeling language, starting with a hierarchically structured model with n-ary relationships as a motivating example in Section II. The representation of this model as a DHHTGraph is introduced in Section III together with the formal definition of its main properties. The specification of DHHTGraph schemas as their metamodels by the modeling language *grUML* is presented in Section IV, and the implementation is described in Section V. Other related approaches on extended graphs are described in Section VI. Finally, Section VII concludes the paper.

II. Motivating example

In *model-driven development*, models of several languages describing different aspects of a system are integrated to one comprehensive system model. Domain-specific modeling languages (DSMLs) may be used to support adequate and natural modeling in certain domains.

Figure 1 shows such an (artificial) DSML depicting the application of a model transformation to integrate a business process model and a feature model to an activity model, including the resulting traceability information. To partially motivate the assumption that enhanced modeling concepts might be useful, this example makes use of some of the cited extensions: hyperedges and nesting of vertices and edges.

Hyperedges: In the example, traceability information is modeled by one comprehensive link connecting all relevant source elements of an applied transformation rule to all relevant target elements. The *traceability link* `t1` with its five ends represents such a dependency between source elements in the

This work is partially funded by the German Research Foundation (DFG), project number EB 119/6-1

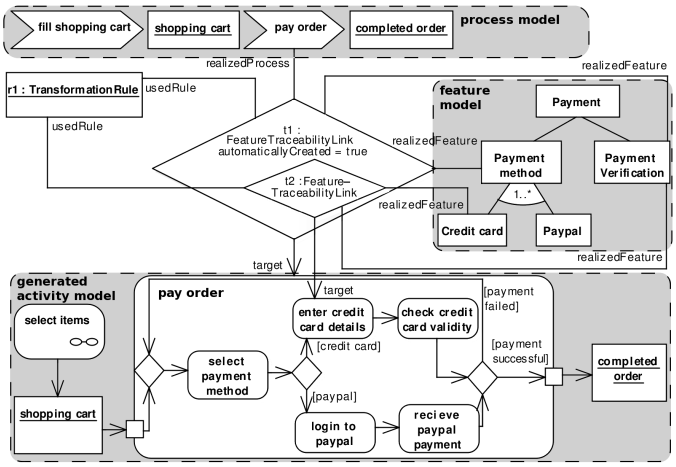

Fig. 1. Example for extended modeling concepts

process and feature models on the one hand and the generated elements in the activity model on the other hand. As its fifth end, the transformation used for generation is attached to the link, as well. The ends of t1 are named, reflecting the different roles of the connected objects in the dependency. There may be multiple ends with the same name at one link, e.g., as shown by the ends named realizedFeature. These names can be interpreted as labels similar to the types of objects and links (see Section IV). The and/or-refinements of features in the feature model can also be seen as 1..n and- or or-hyperedges, respectively, connecting one feature to its children.

Nesting: In Figure 1, the action pay order is refined by its own detailed embedded description inside the activity model. This kind of nesting allows a hierarchical description of activities and is well known from activity descriptions in general. Nesting may also be applied to hyperedges, e.g., the link t1 contains a second link t2, which represents the traceability dependency in a more fine-grained manner. t2 is nested in t1, since it is a refinement. Nesting can be used to improve the comprehensibility of models, if they become large due to a fine-grained modeling.

Level of Detail: Besides the nesting of elements, also the ability to *zoom* in and out of a model can be useful when dealing with large models. Similar to street-maps, where highways are more important than small lanes with many levels in between, there are also elements in software system models which are more important than others, e.g., since they realize major system properties. All elements with the same importance can be seen as one *visibility layer*. The layers can be ordered from the most detailed one to the most abstract one. Sticking to the zoom metaphor, a *visibility view* can be defined, which contains all elements of a given layer and all layers above up to the most abstract one, while all elements of layers below are hidden.

Distribution: A system model usually consists of several independent but interrelated views of the system which are possibly also physically separated and *distributed*, e.g., in the model above the feature tree might be maintained on another site than the process model. However, there are still links which cross boundaries of the parts and there are tasks such as checks for global constraints which need to take the complete system into account. Thus, the complete model needs to be accessible, while keeping the logical and physical separation and the ability to handle each local part independently. In practice, a common repository as e.g. in version management systems is used to represent the complete system in one place. But, the distribution of models should be supported by an implementation of a versatile modeling language.

Conclusion: The practical usage of the enhanced modeling concepts introduced here and their support by modeling tools require an appropriate and algorithmically accessible representation of the models. While *graphs* are widely used as a suitable data structure to represent models (by their abstract syntax graph), they provide no natural counterpart to the enhanced concepts. But, they require to simulate these concepts, e.g., a commonly used workaround for hyperedges and hierarchy is the usage of specifically labeled vertices and edges, which leads to additional exceptional handling of these special elements in all model processing algorithms. Therefore an extension of graphs is required which supplies

- plain graphs, including *Typing*, *Attribution* and ordering of vertices and edges,
- representation of n-ary relationships by *hyperedges*,
- refinement of vertices and edges by *nesting*,
- abstraction levels by *visibility layers*, and
- partitioning of graphs and *distribution* of the parts.

III. Formal definition

While the previous section has described the concepts claimed to be helpful for a natural representation of models, this section presents DHHTGraphs as a formal object realizing this concepts. DHHTGraphs are a generalization of *TGraphs* [1], i.e. typed, attributed, and ordered directed graphs. They provide *hyperedges*, *hierarchy* in the form of nesting and visibility layers and a *distribution* concept, all of which were proposed separately in the literature (see Section VI), but are integrated in one single graph concept here (Figure 2).

The main design decisions allowing this integration were:

- Vertices and edges are dual. They are both typed and attributed, they both may have an arbitrary number of named incidences, and they both may be refined.
- Nested vertices and edges have only vertices or edges on their border, respectively.
- Nesting, visibility layering and partitioning into distributed graphs are compatible.
- Ordering of vertices and edges is consistent with their nesting and their partition into distributed parts.

The formal definition of this graph category is sketched in Figure 2 and is the basis for the coherent and seamless implementation of the single concepts (see Section V). Since the complete formal definition is not in the focus of this paper, we restrict ourselves to the most important constituents of DHHTGraphs, skipping some notational abbreviations and all consistency restrictions. We assume the reader to be familiar

Assume the sets *TypeID* of *types*, *RoleID* of *roles*, *AttrID* of *attribute names*, *Value* of *attribute values*, N of *vertices* and A of *edges* to be given.

$G = (V_{seq}, E_{seq}, \Lambda_{seq}, type, value, \sigma, \kappa, P_{seq}, \pi)$ is a DHHT-Graph iff:

- $V_{seq} \in \mathsf{iseq}\ N$ is a *sequence of vertices*. $V = \mathsf{ran}\ (V_{seq})$ is the vertex-set of G.
- $E_{seq} \in \mathsf{iseq}\ A$ is a *sequence of edges*. $E = \mathsf{ran}\ (E_{seq})$ is the edge-set of G.
- $\Lambda_{seq} : (E \cup V) \to \mathsf{iseq}\ ((V \cup E) \times \{in, out\} \times RoleID)$ is an *incidence sequence*.

Types and attributes:

- $type : V \cup E \cup \{G\} \to TypeID$ is a *typing*, and
- $value : V \cup E \cup \{G\} \to (AttrID \nrightarrow Value)$ is an *attribute assignment* to G and its elements.

Hierarchy:

- $\sigma : V \cup E \cup \{G\} \nrightarrow V \cup E \cup \{G\}$ is an acyclic *nesting function*, and
- $\kappa : V \cup E \cup \{G\} \to \mathbb{N}_0$ is a *visibility function*.

Distribution:

- $P_{seq} = (p_1, \ldots p_n)$ is a *sequence of partial graphs*, and
- $\pi : V \cup E \to P$ is a *partitioning function*.

Fig. 2. Constituents of DHHTGraph

with the common set-theoretic concepts and their notation. While vertices and edges are dual and equitable in all aspects, they are treated as disjoint sets since they represent different concepts of the modeled domain: while vertices represent entities, edges represent any kind of relation among them.

Definition and Notation: Figure 3 visualizes the abstract syntax of the feature model in the upper right corner of Figure 1 as a DHHTGraph to illustrate the mathematically defined properties. The rounded rectangles are vertices while the diamonds are edges. Both kinds of symbols contain the identifier of the respective element, e.g. v1. The arrows between rectangles and diamonds are the *incidences* (the connection points between vertices and edges), whose direction is indicated by an arrowhead.

Vertices and edges are ordered, as it is illustrated by their enumeration notated in parentheses. Similarly, all incidences at a vertex (numbered in square brackets) and at an edge (numbered in curly braces) are *ordered*. As edges are directed and incidences are named, the incidence function Λ_{seq} assigns not only vertices to edges and vice versa but each incidence has a direction (*in* or *out*) and a name out of the set of role names *RoleID*. To assure that the instances of this definition are still graph-like in the sense that there is an explicit distinction between edges and vertices and that there are no edges pointing to edges, a further constraint on Λ_{seq} is defined in the complete definition.

The *typing* of vertices and edges is notated after a colon inside or near the appropriate element and realized by the function *type*, which assigns a type identifier of the predefined set *TypeID* to each vertex and each edge. This set of disjoint

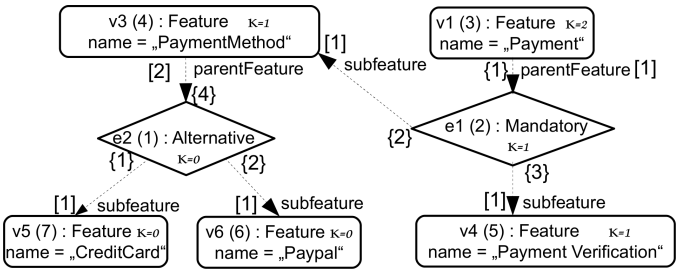

Fig. 3. Example graph of feature model

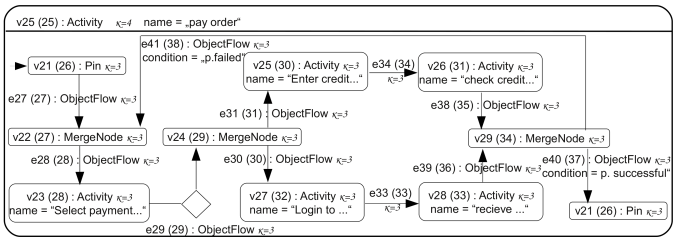

Fig. 4. Example graph of activity model

edge and vertex types is defined by a *graph schema* which is described in the next section. The schema also defines a set of *attribute names AttrID*, on which the attribution function *value* is based. In the example graph, a value for the attribute name is assigned to each vertex.

Hierarchy: Hierarchical structures in graphs are defined by the two functions σ and κ. σ specifies the *nesting* of elements by assigning an element as parent to each vertex or edge. If an element is not nested in another one, its σ value is the graph itself. In Figure 4 this value is expressed by the inclusion of all vertices and edges in the outer vertex v25. The sets of vertices and edges contained in another vertex or edge form a so-called *subordinate graph*, which is itself a DHHTGraph and compatible to the complete graph in all its properties. To assure a well-formed nesting structure, the *border* of subordinate graphs is restricted to contain only elements of the same kind as the refined one, i.e. all elements nested in a vertex or edge x that have connections to elements outside of x have to be also vertices or edges, respectively.

The *visibility layers* are defined by κ. The layers are represented by natural numbers starting with 0 for the most detailed level, which is equivalent to the complete model. To each element a natural number is assigned as the specification of its visibility. Based on these numbers, *visibility views* or *zoom levels* can also be defined as a number, where all elements whose visibility number is higher than or equal to the zoom level number are visible while the elements with lower numbers are hidden. Thus, elements with a low number are visible only in very detailed views while elements with higher κ-values are contained also in more abstract views of the graph. Thus, *Views* on graphs can be defined based on the κ function. These views are also DHHTGraphs according to the definition above. Since complete DHHTGraphs as well as all views and parts of it are defined by the same formalism,

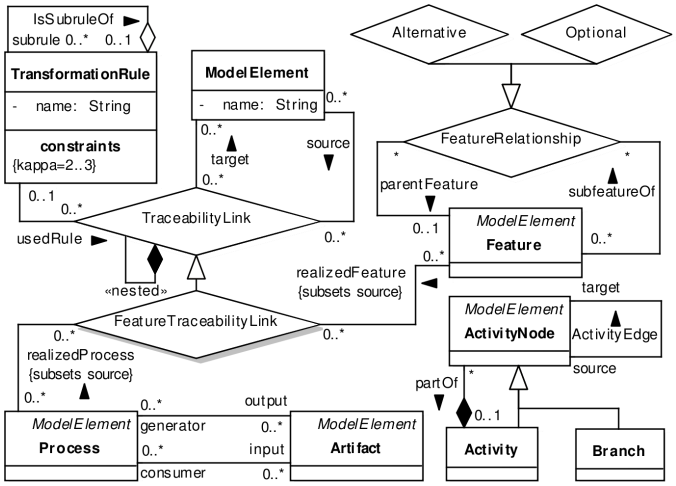

Fig. 5. Example grUML schema for the model from figure 1

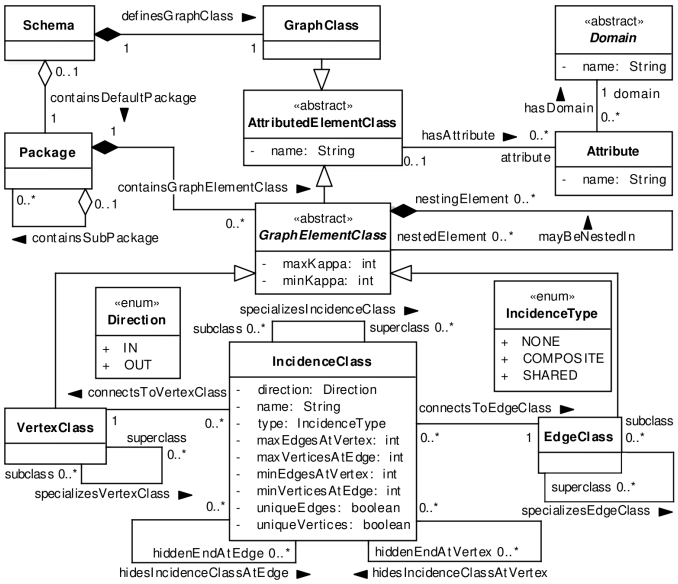

Fig. 6. grUML Metaschema

the same algorithms can be applied to them.

Distribution: Each graph consists of one or more *partial graphs* $p_0 \ldots p_n$ which may be distributed on several independent sites and may be partitioned themselves. Each element is mapped to exactly one partial graph out of the set P by the function π. The properties of each partial graph such as the edge and vertex ordering, the incidences of edges and vertices and their ordering, and the types and attributes of the graph and its elements are compatible to the ones of the complete graph. Regarding the connections between elements in different graphs, there are no further restrictions defined. If only *one partial graph* is considered, all connections to elements outside are clipped and only the connections between elements inside the graph are taken into account. As soon as the *complete graph* is used, all connections are to be considered.

IV. Metamodeling DHHT graph classes

While the formal definition specifies the generic structure of DHHTGraphs, their practical usage as a model representation requires the ability to specify the structure and its constraints in more detail. This leads to the need of metamodeling of DHHTGraphs to describe concrete domain-specific sets of such graphs. As required by the formal definition, some sets TypeID, RoleID, and AttrID are to be given and are used to define typing and attribution of graphs. As the types and attributes used in a graph rely on its application domain, these sets need to be defined in the context of this domain.

In the DHHTGraph approach, *DHHTGraph schemas* are used as *metamodels* to define application-specific sets of DHHTGraphs. These schemas specify the above mentioned sets and functions as well as further features such as connection restrictions and multiplicities. To describe graph schemas, a stereotyped dialect of UML class diagrams named *grUML* is used, allowing the precise and convenient definition of graph schemas using a notation that most modelers are familiar with.

To illustrate grUML, Figure 5 shows a simplified schema for the model depicted in Figure 1. The MOF-metaschema for this language is given in Figure 6 for completeness' sake.

Hyperedges: In Figure 5, vertex types of the language, like TransformationRule and ModelElement, are modeled as classes with attributes. Hyperedge types are notated as *diamonds* like n-ary associations in UML with named and directed ends describing the *incidences* between edges and vertices. As an example, the edge type TraceabilityLink has three incidences named source, target and usedRule. Their direction is depicted by a small triangle. Furthermore, at each incidence there are two kinds of multiplicities (using far notation). As an example, there can be an arbitrary number of TraceabilityLinks at a TransformationRule (indicated by 0..*), while at most one TransformationRule is allowed to be connected as usedRule to a TraceabilityLink. For binary edge classes, which may be notated as binary associations as before, the number of incidences at an edge may be omitted. This is shown e.g. at the edge class IsSubruleOf at the vertex class Transformation. The notation of an association class at an edge class may be used to specify attributes and generalizations for edge classes (not shown).

Vertex- and edge classes may be specialized, as shown exemplarily by TraceabilityLink and FeatureTraceabilityLink. Then, all properties of the superclass such as attributes and incidence classes are inherited. To allow for a detailed and fine-grained specification of possible connections, the inherited incidence classes may be refined using the concepts of subsetting and redefinition as defined in the UML with some extensions for n-ary edge classes. In the example, the subsets keyword is used to refine the inherited incidence class source by the ones named realizedFeature and realizedProcess.

Hierarchy: Besides types of elements and their connections, also the *hierarchy structure* of graphs depends on their application domain and can thus be specified in a graph schema. In Figure 1, the *nesting* of activities was depicted. Using the common UML interpretation, this nesting

relationship would be restricted to classifiers and represented by special links specified by compositions in the metamodel. Since the σ function allows the representation of nesting relationships on vertices and edges similarly, the *composition notation* seems to be suitable and convenient also for the specification of σ in graph schemas. As shown at the edge class TraceabilityLink, a stereotyped composition is used to specify nesting relationships between edge classes.

For the κ *function* as the second part of the hierarchy concept there is no existing notation in UML. In grUML, the generic notation for constraints is used to attach the allowed layers as ranges of numbers to the single elements. An example is shown at the class TransformationRule. For its instances only κ values of 2 and 3 are valid, and thus they are visible on all zoom levels below or equal to 2.

Distribution: Since the concept of partial graphs is not intended to represent aspects of the modeled world, but only to allow the distribution of models on a network (which is not domain-specific and which is independent of the graphs), there are no domain-specific properties of partial graphs which need to be modeled in a graph schema.

V. Implementation

As described above, DHHTGraphs are designed to be used as a flexible data structure in applications. The support of DHHTGraphs as an extended version of the JGraLab [1] Java graph library is currently under development. The main characteristics of this implementation are described in the following. This implementation is derived from the formal definition and provides a full implementation of DHHTGraphs with all their properties. A convenient generic and schema-specific API allows the usage of the formally defined concepts in a natural way.

As far as reasonable, all properties of DHHTGraphs are represented by corresponding Java concepts. Thus, graphs and their elements as well as incidences are realized by plain Java objects. The corresponding classes are shown in Figure 7. The relationships between the elements, like Λ_{seq}, are realized by references between the respective objects, i.e. each incidence object references its connected edge and vertex objects, while the ordering of incidences at edges and vertices is realized as a doubly-linked list. The orderings V_{seq} and E_{seq} are represented similarly. This kind of implementation is based on the *symmetric incidence list* (*SIL*) data structure proposed for plain graphs in [2].

Since the graph and its elements are implemented as Java objects, it is possible to map the typing and labeling information to the respective Java instance. The classes defined in a graph schema are transformed to Java classes extending the implementation classes from Figure 7, and each graph, vertex, edge, and incidence is an object of the class representing its type or label, respectively. Consequently, attributes are represented by Java member fields of the corresponding classes. This implementation allows to handle graph elements as Java objects supporting domain-specific and statically type-safe implementations of DHHTGraph-based software systems.

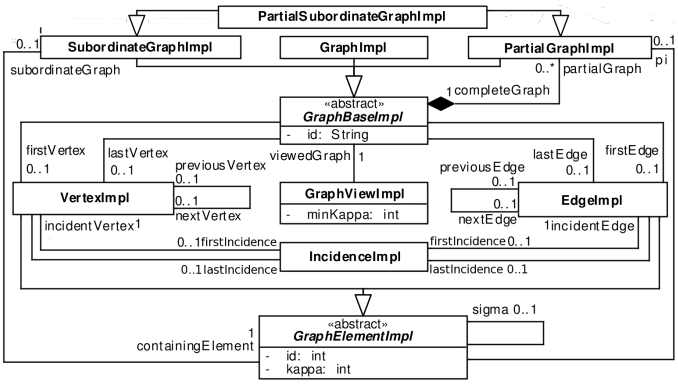

Fig. 7. Main implementation classes

Hierarchy: The *nesting* of graphs in graph elements represented by the σ-function is implemented by a reference named sigma from each element to its parent and an additional graph object for each subordinate graph. As described above, all elements belonging directly or indirectly to the subgraph of an element form a continuous part in V_{seq} and E_{seq}, respectively. Thus, each subgraph contained in an object only needs to know about its first and its last element with respect to the complete graph, since all elements in between also belong to the corresponding subgraph.

To support *visibility layering* the κ value assigned to each element is represented by a plain Java member field kappa of type **int**. A *view* on a graph is thus represented by an own GraphViewImpl-object holding a reference to the viewed graph and the zoom level as its value minKappa. This information suffices to provide full access to the view.

Distribution: The implementation of partial graphs is done analogous to that of subordinate graphs. As the different partial graphs and their elements are located on different sites, some of the references described above can not be realized as plain Java references. Thus, local proxy objects are used as substitutes for the remote objects providing the same interface as the remote elements and delegating all method calls to the substituted object. The Remote Method Invocation (RMI) framework is used to create these proxy objects on demand and allows a convenient usage of distributed graphs.

API: To give an impression of the API provided by the DHHTGraph implementation, Figure 8 shows the Java code of an undirected graph traversal using breadth-first. In addition to the start vertex of the search, an EdgeDirection is passed to the algorithm as a parameter. Depending on the parameter value (IN, OUT or INOUT), either a directed forward, a directed backward, or an undirected traversal is executed without any further change necessary. The third parameter specifies the subgraph on which the search should be performed. As all kinds of graphs and subgraphs implement the Graph-interface, this enables their flexible and uniform usage.

VI. Related work

Many Extensions of the concept of graphs have been proposed by researchers. The most relevant for the development

```
void bfs(Vertex start, EdgeDirection dir, Graph graph) {
 GraphMarker marker = new BooleanGraphMarker(graph);
 Queue<Vertex> queue = new LinkedList<Vertex>();
 marker.mark(start); queue.offer(start);
 while (!buffer.isEmpty()) {
  Vertex currentV = queue.poll();
  for (Incidence curIncAtV : currentV.incidences(dir, graph)) {
   Edge currentE = curIncAtV.getEdge();
   if (!marker.isMarked(currentE)) {
    marker.mark(currentE);
    for (Incidence curIncAtE: currentE.incidences(dir,graph)) {
     Vertex other = curIncAtE.getVertex();
     if (!marker.isMarked(other)) {
      marker.mark(other); queue.offer(other);
} } } } } }
```

Fig. 8. Breadth-first search on DHHTGraphs

of DHHTGraphs are shortly described in the following.

Hyperedges: Several approaches for hyperedges have been developed, from simple undirected ones as defined by Berge [3] to complex ones as defined by Habel [4]. While in the latter the incidences of a hyperedge are ordered, the approach does neither allow for the labeling of incidences nor for specifying incidence classes. A basic form of labeling is supported by the Graph eXchange Language (GXL) [5] but without a refinement concept on incidence classes.

The usage of *hyperedges* and their value has been discussed controversially. Engels and Schürr [6] propose to keep links a simple as possible. They argue, that otherwise the separation of edges and vertices vanishes, such that hyperedges become vertices and their ends become a new kind of plain edges.

Hierarchy: The different approaches proposed in the area of *hierarchical structuring of graphs* can be divided into *three categories*, based on their fundamental ideas.

The probably most obvious kind is the *refinement of nodes and edges by nested graphs*. Whereas some approaches do not allow for connections between a nested graph and elements outside, this is possible in Harel's higraphs [7] and in the hierarchical graphs defined by Engels and Schürr [6]. While only the first one also supports hyperedges, the second one allows the description of graphs including nesting structure by graph schemas. However, both approaches do not support the nesting of graphs in edges, which again is possible in GXL.

Visibility layering is realized only for plain graphs without hyperedges in other approaches. Buchholz [8] presents this kind of hierarchy in detail with the variant, that derived edges on higher levels represent the hidden reachability information of lower levels, whereas in DHHTGraphs, the nesting of graphs in edges used to represent such hidden information.

The third kind of hierarchy is the *replacement* of elements by graphs. The concept proposed by Langou and Mainguenaud [9] allows the replacement of vertices and edges by graphs but is limited to binary edges, other concepts such as the one proposed by Habel [4] support hyperedges, but are restricted on the replacement of either vertices or edges. These concepts are the basis for graph-grammars and some graph transformation systems, but they seem to be of limited use for the representation of models.

Distribution: A simple notion of distributed graphs is possible with RDF [10]. The RDF data structure describes plain directed graphs, which are represented as sets of triples. Each triple represents a binary edge with its type (property) and its start and end vertex. By using URIs as edge endpoints, RDF supports links between vertices in distributed graphs. However, the coupling is only loose.

A more convenient realization of distributed graphs is available in DRAGOS [11]. Vertices in one graph may be proxies for vertices in another graph, referring to them using URIs as unique identifier. In contrast to DHHTGraphs, there is no notion of a complete graph and a distribution of edges is not possible. A concept of distributed TGraphs also based on proxy vertices has been implemented in JKogge [12] where parts of a graph can be distributed over several sites.

VII. Conclusion and further work

DHHTGraphs are a powerful and flexible data structure for the representation of complex models. The support of hyperedges with named ends allows a natural and concise representation of complex n-ary relationships, while the two hierarchy concepts (nesting and visibility layers) enable abstraction and different views on a system. The distribution of partial graphs over several sites and the choice to use them independently or as one complete graph enables a convenient support of working with distributed data.

This paper introduced DHHTGraphs as a comprehensive, integrated concept by presenting their formal definition and a UML-like metamodeling language. The implementation of DHHTGraphs as an extension of the JGraLab graph library exists as a prototype.

References

[1] J. Ebert, V. Riediger, and A. Winter, "Graph Technology in Reverse Engineering, The TGraph Approach," in *10th Works. Softw. Reeng.*, R. Gimnich *et al.*, Eds., vol. 126. Bonn: GI, 2008, pp. 67–81.

[2] J. Ebert, "A Versatile Data Structure For Edge-Oriented Graph Algorithms," *Communications ACM*, vol. 30, no. 6, pp. 513–519, 6 1987.

[3] C. Berge, *Graphs and Hypergraphs*. Elsevier Science Ltd, 1985.

[4] A. Habel, "Hyperedge Replacement Graph Grammars," *Lecture Notes in Computer Science*, no. 643, 1992.

[5] R. C. Holt *et al.*, "GXL: a graph-based standard exchange format for reengineering," *Sci. of Comp. Prog.*, vol. 60, no. 2, pp. 149–170, 4 2006.

[6] G. Engels and A. Schuerr, "Encapsulated Hierachical Graphs, Graph Types, and Meta Types," in *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Works. on Graph Rew. and Comp.*, vol. 2. 1995.

[7] D. Harel, "On visual formalisms," *Communications of the ACM*, vol. 31, no. 5, pp. 514–529, May 1988.

[8] F. Buchholz, "Hierarchische Graphen zur Wegesuche," Ph.D. dissertation, Universät Stuttgart, 2000.

[9] B. Langou and M. Mainguenaud, "Graph Data Model Operations for Network Facilities in a Geographical Information System," in *6th Int. Symp. on Spatial Data Handl.*, vol. 2, Edinburgh, 1994, pp. 1002–1019.

[10] G. Klyne and J. J. Carroll, "RDF concepts and syntax specification," W3C, Tech. Rep., 2 2004, http://www.w3.org/TR/rdf-concepts.

[11] B. Böhlen, *Ein parametrisierbares Graph-Datenbanksystem für Entwicklungswerkzeuge*. Aachen: Shaker Verlag, 12 2006.

[12] J. Ebert, R. Süttenbach, and I. Uhe, "JKogge: a Component-Based Approach for Tools in the Internet," in *Proc. STJA '99*, Erfurt, 1999.