Software Evolution
via
Hierarchical Hypergraphs with Flexible Coverage

WOLFRAM KAHL
Institut für Softwaretechnologie, Fakultät für Informatik
Universität der Bundeswehr München, D-85577 Neubiberg
kahl@ist.unibw-muenchen.de

Abstract

We present a simple, abstract approach to the use of hierarchical hypergraphs in software evolution. Borrowing ideas from graph transformation and attribute grammars, we show how these hypergraphs can be used in a flexible way to cover *all or part of a software development process.*

This unifying framework allows to design a set of tools *based on common data structures and representations and applicable to diverse tasks and settings.*

1 Introduction

When a piece of "software" evolves with *full* formal support, this implies that for all components that belong to that piece of software, such as

- domain knowledge (ideally in the shape of formal theories),
- requirements,
- user documentation,
- design decisions and their motivations,
- design documentation, and
- "source code",

we have the following:

- the formal support is aware of these components and their structure, and
- the formal support is aware of all kinds of relations between these components, down to arbitrary constituent levels.

Furthermore, to support a more abstract view on evolution, we need that

- the formal support is aware of different versions of the system and of the relations between them.

Therefore, to the formal support, the whole system together with its history and variations appears as a single, though highly structured, *hypergraph*.

However, it will be rare to have full formal support for the whole of the software development process. Usually, already the availability of semi-formal approximations to the full hypergraph of formal support will be considered as an improvement in the process. Furthermore, sometimes formal support needs to be added only later in the process, on an existing system, for example in re-engineering projects. There, it will usually not be possible to derive the whole hypergraph from a given system state (repository) without expensive human interaction, since frequently the relations that have to be represented in the hypergraph are not obvious from the system state. For example, it may be (formally) undecidable which requirements specification or domain knowledge formula is reflected in a specific design decision. Therefore,

- this hypergraph will be *incrementally constructed* as one activity of the development process among others, and
- this hypergraph will have to be *maintained* along with the system representation, or better,
- this hypergraph will have to be *viewed as being* the system representation, even if many edges are "missing".

Tools that aid maintenance of such a hypergraph would be easier to implement if no part of the whole system representation could be changed without awareness of the impact on

the hypergraph structure. That approach, however, would imply almost zero interoperability, and may also be a serious impediment to scalability.

External tools, and, to a certain degree, distributed development will always at least locally and temporarily destroy the hypergraph structure.

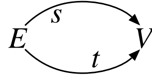
In order to deal with all these facets of software evolution reality, we propose a formalism of *hierarchical hypergraphs with flexible coverage*.

2 Hierarchical Hypergraphs with Flexible Coverage

We propose a hypergraph formalism using hierarchical hypergraphs along the lines of the "higraphs" of Tourlas [1]. Therefore, we first introduce these "higraphs", and then explain how we instantiate this definition for our purposes.

There are many ways to approach the definition of graphs and hypergraphs, and also many ways to specify graph transformations. Because of the high level of abstraction and generality, approaches based in category theory are very prominent, and the basic techniques of the caategorical approach to graph transformation are well-established and accepted.

In category theory, there is one particularly simple and useful approach to what turns out to be a very conventional definition of graphs: One starts by defining a category $\mathbf{G}$ with two objects and two non-identical morphisms, postulationg only the category equations:

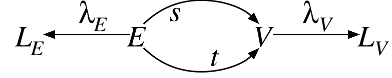


A graph is then a functor from this category $\mathbf{G}$ into the category *Set* of sets and total functions between sets. This means, that for every graph G_i there are

- a vertex set V_i,
- an edge set E_i,
- a total function $s_i : E_i \to V_i$, which is understood to associate every edge with its *source* vertex, and
- a total function $t_i : E_i \to V_i$ which associates every edge with its *target* vertex.

From the definitions of categories and functors one obtains a natural definition of graph homomorphisms (as *natural transformations* between functors). General theory about *set*-valued functors then immediately produces a wealth of results about this category of graphs.

Vertex and edge labellings may be added by extending the base category with additional objects for the label sets, and labelling morphisms, thus obtaining a category $\mathbf{GL}$:



All this is well-established.

Now we come to the idea behind the "higraphs" of Tourlas [1], which is in fact extremely simple: Use the above setting with base category $\mathbf{G}$, but replace the category *Set* of sets and total functions between sets with the category *PO* of partially-ordered sets and order homomorphisms (i.e., monotonic total functions) between them.

Thus, every higraph consists of the same four components as a graph, but the source and target functions now have to be monotonic: whenever two edges $e_1, e_2 : E_0$ in a higraph H_0 are related by the edge ordering, i.e., when we have $e_1 \leq_E e_2$, then the incident vertices have to be related by the vertex ordering, i.e., we also need to have $s_0(e_1) \leq_V s_0(e_2)$ and $t_0(e_1) \leq_V t_0(e_2)$.

Tourlas uses these graphs most notably for representing statecharts, with their hierarchical state transition diagrams and their edges at and between different levels in the hierarchy.

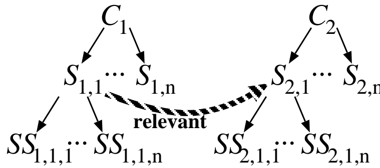
This approach even carries over to the labelled case without problems: We then just need order-preserving labelling functions. A particularly simple instance might, for example, use the trivial identity ordering on the edge label set; this then implies that whenever $e_1 \leq_E e_2$, then their labels conincide: $\lambda_E(e_1) = \lambda_E(e_2)$.

This already shows that this approach to hierarchical graphs is quite flexible. We now propose a hypergraph formalism which is an instance of edge-labelled higraphs in the following way:

- Nodes represent basic items of the system representation, such as formulae, natural language sentences, source code statements.
- Subsystems are, for simplicity, considered as the sets of nodes they contain. Such subsystems are going to be used as the vertices of our higraphs.
- Edges will be hyper-edges with a non-zero number of tentacles attached to them; instead of just two tentacle rôles "source" and "target" we admit an arbitrary number of tentacle rôles, and correspondingly expand the number of morphisms between edge set and vertex set. Since these tentacle rôles will have to be interpreted as *total* monotonic functions, we may choses empty subsustems as targets for rôles where these rôles are not applicable to the edge in question..
- Edges will be labelled, sometimes just with rôles, sometimes e.g. with in addition formal proofs that establish the relation asserted by the edge.

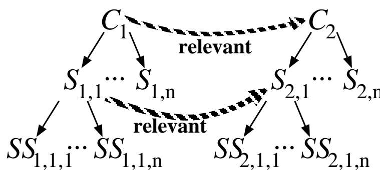
On these hyper-graphs, transformations and transitions are defined as follows:

- *Covering transitions* add edges in a way that roughly corresponds to calculation of attributes in attribute grammars. Consider the following as an example: Assume that a certain section of the specification has a "relevance" edge to a certain section of the user documentation:



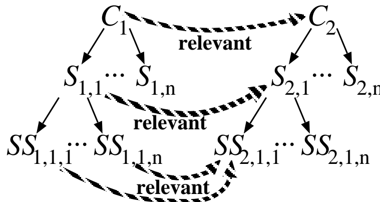
  then:

  - A "relevance" edge will be introduced between the chapters containing those sections:



    This may happen automatically.

  - Edges may be introduced between subsections or formulae in the specification and subsections in the user documentation:



    This may need human assistance.

- *Transfer transitions* add edges in a similar way to relate new versions of parts of the system with the rest of the system. Part of this may be automated, and human assistance may be needed in certain cases as in most version management systems.

- *Lossy transitions* remove edges in response to changes to a part of the system that cannot be assured to have preserved the properties represented by those edges. For example, manual editing of any document will in many cases destroy (or mark as unreliable) most edges incident with that document.

- *Transformations* change the structure of some part of the system, and may add and delete nodes and edges. (The fact that some part of the structure is derived from a transformation will usually be recorded in appropriate edges resulting in "self-covering" transformations.) Transformations can serve the most diverse purposes, and higher degree of formality in the development will usually involve a higher percentage of transformations in the process.

Formally, transitions will usually be described by total single-pushout rules, while transformations will essentially be conventional double-pushout rules.

On such a hypergraph representing a system state, several predicates will be defined, such as:

P1: the hypergraph covers the whole system representation (no covering or transfer transitions can be applied)

P2: the hypergraph covers the whole representation of a specific version

P3: the hypergraph covers all relations between two specific versions

P4: the hypergraph demonstrates that, in a specific version, a specific set of requirements is fulfilled by the implementation

P5: the hypergraph demonstrates that, in a specific version, a specific set of requirements is reflected in the user documentation

...

3 How and why can this formalism be used to provide tool support for evolution?

As documented by the examples given above, a hierarchical hypergraph is a universal framework that can be used to represent and document very different kinds of relations between very different parts and aspects of the system. Some parts of the system, e.g. UML diagrams or finite-state machines, may even be directly encoded as sub-hypergraphs in the same formalism.

The fact that a single formal model stands behind all aspects of the system structure makes it easy to develop a coherent tool set of tools containing special functionality for special aspects of the system, or for special aspects for the interaction with the hypergraph structure:

- Visualisation may be unified, and will automatically be available at all levels of the hierarchy.

- Closure tools will have different derivation components for correctness proofs than for documentation coverage checks.

- Derivation tools may have different instances for different kinds of diagrams and different target paradigms.

Although a unified approach is taken, there is no necessity to use a unique tool, as long as the different tools operate on the same formal model and with compatible representations.

Since not all of the desirable predicates (e.g., **P1**) need to hold all the time during development, tools cannot rely on such assumptions, either, so there is a certain *built-in robustness* in our approach. In particular the possibility to have parts of the system loosing their connections with the rest of the system, or starting their existence in such an isolated state, is the key to interoperability with other tools that are not aware of the hypergraph structure, but only operate on certain (sets of) nodes. Some external tools may still provide some certain kinds of to relevant structure; this can then be used by hypergraph tools e.g. to automate at least certain coverage processes.

4 For which aspects of software evolution can this formalism provide support?

Since our formalism is essentially a meta-formalism, it can be used for all kinds of software evolution and in all parts of the software development process as long as tools with the relevant additional capabilities are available.

It is of course possible to encode even the formulae of the requirements specification as hypergraphs, and similarly the "source-code" of software products, and have hypergraph transformations for the complete development, proof, and maintenance process. However, this will probably be the exception.

More or less at the other extreme, it is also conceivable that a re-engineering project starts out with just nodes, namely the existing source code and documentation, and progressively adds edges as relations between documentation and source code are discovered, and adds nodes as new documents are added.

5 Items for Discussion

Instead of a conclusion, let us raise a few points that might deserve discussion:

- In our examples, edges range from the "soft", such as documentation coverage, to the "hard", such as documenting transformation steps and formal correctness proofs. I would consider this as an advantage, since it gives users flexibility with respect to the degree of formal support they wish to see integrated into their process. Since "hard" edges are usually accessible to automatic proof-checking tools, predicates asserting consistency of proof-carrying subgraphs may be defined and checked automatically.

  Would a more rigorous support of consistency blend in equally well with a potentially mixed environment?

- In the implementation of tools for our hypergraphs, edges will exist outside the linked documents, employing addressing mechanisms such as e.g. XLink. Are there other obvious candidates for standardised representations?

- Fine-grained distributed locking will be necessary to minimise conflicts between concurrent application of hypergraph-aware tools — is this considered problematic?

- We mentioned the possibility to store formal correctness proofs in edges (a variant would be to store them as nodes and just link them via edges) — would other ways of linking in external theorem provers be more attractive?

References

[1] K. Tourlas. Towards the principled design of diagrams in computing and software engineering. slides from a talk given in Birmingham on 27th October 2000, Oct. 2000. URL: http://www.dcs.ed.ac.uk/home/kxt/birmingham_4up.ps.gz.