Formalizing Architectural Refactorings as Graph Transformation Systems

Lars Grunske
Boeing Postdoctoral Research Fellow, School of ITEE University of Queensland, St Lucia, Brisbane, Australia
Email: grunske@itee.uq.edu.au

Abstract

Architectural refactorings are an appropriate technique for the development and improvement of architectural specifications. However, these refactorings are often applied manually. This paper presents a mapping of an architectural specification language to a hypergraph-based data structure. Thus, architectural refactorings can be formalized as hypergraph transformation rules and can be applied automatically.

1. Introduction

An intrinsic property of software in a real-world environment is its need to evolve [20]. In recent years, it has been widely acknowledged as a good practice to divide this evolution into two distinct steps [23]:
1. Software Evolution without introducing any new behavior on the conceptual level and only restructure the software design to improve quality factors such as maintainability, etc. or
2. Software Evolution to extend the design and modify the software behavior.

This first step has been called refactoring [21], and is now seen as an essential activity during software development and maintenance [8].

By definition, refactorings should be behavior-preserving transformations of code- and design specifications. However, refactorings could be also applied to architectural specifications. An example of an architectural refactoring is the triple modular redundancy refactoring (TMR-refactoring) [5], which uses three copies of a component in combination with a two-out-of-three voter to reduce the possibility that a failure of a component affects the correct behavior of the system. An application of this architecture refactoring is necessary if a failure of a component (e.g. a sensor) can lead to dangerous system behavior.

The problem with architectural refactorings in industrial projects is that they are often applied manually [4]. This paper presents an approach based on graph transformation systems for automatic application of architectural refactorings. For this purpose, an architecture description language is mapped to a hypergraph based data structure that enables to specify architecture transformation with hypergraph transformation rules. The benefits of hypergraphs are the ability to describe hierarchical structures [11], which is especially important for architecture specifications and improves the approach of Mens, Janssens and Demeyer [19].

The remainder of this paper is organized as follows: In section 2, an introduction to hypergraph theory is given. Section 3 defines a mapping between hierarchical typed hypergraphs and the architecture description language COOL (Component-based Object Oriented Language [16]). Section 4 presents the basic concepts of hypergraph transformation rules, which are applied in section 5 for specification of architectural refactorings (homogeneous redundancy pattern). Finally, section 6 contains concluding remarks and points out directions for future work.

2. Theory of Hierarchical Hypergraphs

Hypergraphs are a generalization of normal graphs, where an edge can be associated to more than two nodes. They are a data structure that can be applied in many areas [9].

2.1 Typed Hypergraphs

According to [9] a typed hypergraph can be defined as follows:

Definition 2.1 Typed Hypergraph Let \( \nu \) be a set of node types and \( \nu_n \) be a set of node types, then a hypergraph \( G \) from the possible set of hypergraphs \( G \) over \( \nu \) and \( \nu_n \) is characterized by the tuple \( \langle \nu, E, att, lab \rangle \), with two finite sets \( V \) and \( E \) of nodes (or vertices) and hyperedge, a labeling function \( \text{lab}: V \rightarrow \nu \cup E \rightarrow \nu_n \) and an attachment function \( \text{att}: E \rightarrow V' \), where \( V' \) denotes a sequence of nodes with a specified order. As the symbol for the addition and subtraction on this sequences we use \( \oplus \) and \( \ominus \).

Proceedings of the Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN’05)

0-7695-2294-7/05 $20.00 © 2005 IEEE

Authorized licensed use limited to: SWINBURNE UNIV OF TECHNOLOGY. Downloaded on August 11,2010 at 10:19:01 UTC from IEEE Xplore. Restrictions apply.
The labeling function allocates for each hyperedge and each node a hyperedge or node type and the attachment function \( att \) assigns a sequence of nodes to a hyperedge. The number of elements in this sequence \( |att(e)| \) is called the arity of the hyperedge.

### 2.2 Hierarchical Hypergraphs

Typed hypergraphs, as presented in Definition 2.1, have a flat structure. For the mapping of software architecture to hypergraphs, it is necessary to structure the hypergraphs hierarchically. This leads to a reduction of the complexity and introduces abstraction concepts to hypergraphs.

Hierarchical hypergraphs as introduced in [11] use refinement of special hyperedges. These hyperedges are used to embed a multi-pointed hypergraph, which contains a set of external nodes. These multi-pointed hypergraph are defined as follows:

**Definition 2.2 Multi-Pointed Hypergraph** A multi-pointed hypergraph is characterized by the tuple \( \langle V, E, att, lab, ext \rangle \), where \( \langle V, E, att, lab \rangle \) is a typed hypergraph and \( ext \) describes a sequence of external nodes \( ext \in V^* \).

The arity of a multi-pointed hypergraph can be determined by the length of the external node sequence \( |ext| \). For the embedding of a multi-pointed hypergraph in another graph a hyperedge with the same arity can be refined. This hyperedge must be removed from the graph and the multi-pointed hypergraph is included in the remaining hyperedge frame. This hyperedge frame consists only of the associated nodes of the removed edge. The nodes of the hyperedge frame and the external nodes are mapped mutually and they define the glue between the enclosing graph and the embedded hypergraph [11].

Based on hyperedge refinement and the definition of a multi-pointed hypergraph a hierarchical typed hypergraph can be defined as follows:

**Definition 2.3 Hierarchical Typed Hypergraph** A hierarchical typed hypergraph \( G \) from the set of hypergraphs \( G \) over \( L_v \) and \( L_E \) is characterized by the tuple \( \langle V, E, att, lab, ext, cts \rangle \), where \( \langle V, E, att, lab, ext \rangle \) is a multi-pointed hypergraph and \( cts : E \rightarrow G \) is an assignment function which assigns contained hierarchical typed hypergraphs to a hyperedge.

Due to the recursion of this definition, the structure of a hypergraph is defined inductively over levels of the hierarchy. In the lowest level \( G_0 \) no hyperedge \( e \in E \) contains an embedded hypergraph \( cts(e) = \emptyset \). Thus, all hypergraphs in \( G_0 \) are regular typed graphs. In higher hierarchy levels \( G_n \), with the function \( cts : E \rightarrow G \) the embedded graphs are assigned to the hyperedges.

Based on object-oriented concepts the node and hyperedge types can be specified as classes. These classes can contain a set of application specific attributes and operations. Furthermore, hypergraphs can be also typed as classes. This leads to the meta-model of typed hierarchical hypergraph in figure 1.

![Figure 1. Meta-model of a typed hierarchical hypergraph](image)

### 3. Structure Specification with Typed Hierarchical Hypergraphs

#### 3.1 Software Architecture

To realize a hypergraph based structure specification the meta-model of typed hierarchical hypergraphs must be extended. This is presented in figure 2 by introducing of a new meta-level. This meta-level contains the relevant elements to specify a software architecture in the architecture description language COOL (Component based Object Oriented Language, see [14] and [15] for more details).

The elements in the new meta-level are divided into base-classes and architecture-classes. The baseclasses are needed for the structuring of the meta-model and are directly derived from the hypergraphclasses. The architecture-classes are derived from the base-classes. They are shaded gray in figure 2. From these base-classes, the elements of concrete software architectures must be derived. Thus, they define the vocabulary of the architectural description language (ADL) [17].

The fundamental architecture-classes are the component- and connector-class. They are derived from the meta-class software element, which is derived from the meta-class hypergraph. The structure specification is consequently a component-connector-model [9]. The components and connectors interact with the environment via interface elements, which are typed by the meta-class port and role. These meta-classes are derived from the meta-class node in hypergraph specification. Every software element contains a finite set of these interface elements, because of the composition \( V \) in the hypergraph-meta-level. The hypergraph based structure specification of the ADL COOL distinguishes between flat and hierarchical software elements. A flat software element contains only one hyperedge of the type module. This hyperedge is associated to all contained nodes of the software element.
Hierarchical software elements can contain a set of hyperedges and some of them can be complex hyperedges that can contain other software elements. Thus, components and connectors can be decomposed hierarchically. Consequently, a composition hierarchy represents the containment of the software elements as a tree. The leaf nodes of this tree are flat software elements. The root is a special component called genesis component. Based on the genesis component, the system can be instantiated recursively by the compositions $V$ and $E$ of the hypergraph meta-class.

The communication connections between two software elements are modeled by the meta-classes connection and binding. These are derived from the meta-class hyperedge and are connect to two or more interface elements. The difference between the connection and the binding mechanism is that a connection interconnects two interface elements on the same hierarchical level, whereas a binding interconnects two interface elements in different hierarchical levels.

### 3.2 Hardware Architecture

For the complete specification of system architectures the meta model should be extended by hardware aspects [13]. This extension is presented in figure 3. The following four hardware types are added to the meta-model of the ADL COOL: sensor, actuator, hardware platform, hardware channel.

The sensors and actuators are input / output devices and represent the interface to the system environment. They can be modeled as message sinks (actuators) and sources (sensors). The hardware platforms are needed to process the computational activities. Thus, each flat software element is associated to one hardware platform and in the other direction to each hardware platform a set of basic software elements is assigned. A hardware channel connects two hardware platforms and represents the physical transmission medium and the communication mechanism. For a hypergraph-based specification of these hardware elements the COOL-meta-model is extended by the introduction of new meta-classes for the four hardware types as presented in figure 3. These meta-classes are derived from the meta-class hypergraph. Thus, hierarchical decompositions of hardware elements are possible. For the specification of the processing relation, the meta-class processing-node is derived from the meta-class node and the meta-class processing-connection is derived from the meta-class hyperedge.
3.3 Graphical Notation

Architecture specifications can be very complex [12]. Thus, a graphical notation is desirable to handle this complexity. Due to the popularity in the architectural community this notation uses architecture specific graphical symbols [21] instead of hypergraph specific symbols. These graphical symbols are presented in table 2.

<table>
<thead>
<tr>
<th>Hypergraph</th>
<th>Architectural Element</th>
<th>Graphical Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypergraph</td>
<td>Component, Sensor, Actuator, Hardware-platform</td>
<td></td>
</tr>
<tr>
<td>Hypergraph</td>
<td>Connector, Hardware-cannel</td>
<td></td>
</tr>
<tr>
<td>Node</td>
<td>Port</td>
<td>![Port Symbol]</td>
</tr>
<tr>
<td>Node</td>
<td>Role</td>
<td>![Role Symbol]</td>
</tr>
<tr>
<td>Hyperedge</td>
<td>Connection, Binding, Processing-Connection</td>
<td></td>
</tr>
</tbody>
</table>

4. Graph Transformation System

4.1 Hypergraph Replacement for Typed Hypergraphs

A hypergraph transformation rule defines in an abstract manner the replacement of a subhypergraph in an application graph. The connection between the replacing hypergraph and the application graph is done by the gluing approach [7] and a hypergraph transformation rule can be formally defined as follows [6]:

**Definition 4.1 Hypergraph transformation rule** A hypergraph transformation rule is a tuple \((G_i, G_j, G_k, l, r)\), with three hypergraphs \(G_i, G_j, G_k \in \mathcal{H}\) called left-hand-side, right-hand-side graph and interface hypergraph and two hypergraph morphisms \(l: G_i \rightarrow G_j\) and \(r: G_j \rightarrow G_k\). For simplification a hypergraph transformation rule can be denoted as \(G_i \leftarrow G_j \rightarrow G_k\).

For the implementation of hypergraph transformation rules the algebraic approach [7] is favored [6]. This approach is based on the construction of pushouts and pushout complements in the category of typed graphs. To visualize a graph transformation rule with the algebraic approach the following pushout diagram can be used:

\[
\begin{array}{ccc}
G_i & \xrightarrow{l'} & G_j \\
\downarrow & & \downarrow \\
\downarrow & & \downarrow \\
G & \leftarrow & D \rightarrow G' \\
\end{array}
\]

The context graph \(D\) and the morphism \(l'\) are identified by constructing a pushout complement of the \((G_i, o, l)\). Afterwards the resulting graph \(G'\) can be constructed with the pushout of the tuple \((G_i, o', r)\). To illustrate the approaches in figure 4 an example is presented. The upper part of this example contains the graph transformation rule and in the lower part the application \(G\), context \(D\) and resulting graph \(G'\) are presented.

![Example of Hypergraph Transformation Rule](image)

**Figure 4. Application of graph transformation rules**

4.2 Hypergraph Replacement in Hierarchical Typed Hypergraphs

For the hypergraph replacement in hierarchical typed hypergraphs first of all morphisms must be introduced in this category. This can be defined according to [6] inductively over the embedding hierarchies:

**Definition 4.2 Morphism in hierarchical typed hypergraphs** Let \(G = \{V, E, att, lab, cts\}\) and \(G' = \{V', E', att', lab', cts'\}\) be two hierarchical typed hypergraphs, then a morphism \(m: G \rightarrow G'\) is defined by a tuple \((m_v, m_e, M)\), where \((m_v, m_e)\) characterizes a morphism of a flat graph and \(M\) is a family of morphisms \(M_v\) for the embedded hypergraphs of all complex hyperedges \(e \in \text{dom}(cts)\). Thereby each morphism \(M_v\) can be defined as follows: \(M_v: cts(e) \rightarrow cts'(m_v(e))\).

In [11] Hoffman presents that in the category of hierarchical typed hypergraphs pushouts and pushout complements can be constructed. Thus, the application of hypergraph transformation rules in hierarchical typed hypergraphs can be applied similar to the algebraic approach typed hypergraphs.

4.3 Graphical Representation

For the representation of complex graph transformation rules in [16] a graphical notation is proposed. This notion is called \(\Delta\)-notation due to its \(\Delta\)-shaped. In the \(\Delta\)-notation the interface graph is specified within the delta. The graph elements of the left-hand-side and right-hand-side graph are displayed on the left and right side of the \(\Delta\). In addition to the graph elements of the left-hand-side, right-hand-side and interface graphs in the \(\Delta\)-notation an optional context specifies additional required graph elements. These are noted with an \(\ast\), in case they do not belong to the interface graph.
Furthermore, the required context is specified by positive and negative application conditions [10]. These are located above and below the delta.

**Figure 5. The Δ-notation**

An example of a graph transformation rule in the Δ-notation is given in figure 6. This example uses the already introduced rule (see figure 4).

**Figure 6. Example of a graph transformation rule in the Δ-notation**

### 5. Example

In this section with the homogeneous redundancy refactoring [5] (or triple modular redundancy pattern) an often used refactoring is specified as a hypergraph transformation rule to show the technical feasibility. The objective of this refactoring is to increase the safety of a system, by use multiple homogeneous software components operating in parallel on different hardware platforms and a comparator that checks the computational results by a majority voting (often a 2-of-3 voting is used). This structure detects random and single point failures of the hardware platforms. Addition the system’s reliability is improved, if the comparator’s hardware platform is more reliable than the other hardware platforms.

The transformation operator for the homogeneous redundancy refactoring can be specified with two hypergraph transformation rules. The first rule (figure 7) only considers the software aspects and multiplies a software component three times. All of them are processed on the same hardware platform.

**Figure 7. Transformation operator to multiply software components and introduce a comparator**

Due to the application condition, this rule can only be applied, if a component \( x_1 \) is identified, which does not meet its reliability requirements. If this application condition is fulfilled, the component \( x_1 \) can be replaced by the right side of the delta. The ports inside the delta are remained unaffected. Thus, they can be seen as gluing points between the rest of the architecture and the new architectural elements. For a complete specification of the homogeneous redundancy refactoring, the software components must be distributed to different hardware platforms. A suitable hypergraph transformation rule to distribute software components is presented in figure 8. This rule deletes the left processing connection and substitutes it with the right one. Thus, the component is now processed on another hardware platform.

**Figure 8. Transformation operator to distributed and reallocate software components**

### 6. Conclusion and Future Work

In this paper, the mapping of the architecture description language COOL [15] to a hierarchical typed hypergraph was presented. Based on this mapping architectural refactorings could be formalized as hypergraph transformation rules. These hypergraph transformation rules could be automatically applied to architecture specifications. The technical feasibility of a behavior preserving architectural transformation is given by the homogeneous redundancy pattern, which is specified by hypergraph transformation rules.
Due to the common specification formalisms between COOL and CSD (composite structure diagrams in the UML 2.0) a generalization of the presented concepts could also be applied in the UML [2]. We want to explore the details in future work.

Additionally, based on these results further architectural refactorings, i.e. the heterogeneous-redundancy-, the watchdog-, the monitor actuator- [5] and the task-clustering-pattern [13] can be mapped to hypergraph transformation rules. This leads to an appropriate hypergraph transformation system which can be used automatically to improve software architectures.

10. References