Software Architecture Transformations

Hoda Fahmy
Dep't. of Computer Science
University of Toronto
fahmyh@cs.toronto.edu

Richard C. Holt
Dep't. of Computer Science
University of Waterloo
holt@plg.math.uwaterloo.ca

Abstract

In order to understand and improve software, we commonly examine and manipulate its architecture. For example, we may want to examine the architecture at different levels of abstraction or zoom-in on one portion of the system. We may discover that the extracted architecture has deviated from our mental model of the software and hence we may want to repair it. This paper identifies the commonality between these architectural transformation actions – that is, by manipulating the architecture in order to understand, analyze, and modify the software structure, we are in fact performing graph transformations. We categorize useful architectural transformations and describe them within the framework of graph transformations. By describing them in a unified way, we gain a better understanding of the transformations and thus, can work towards modeling, specifying and automating them.

Keywords: *software architecture, graph transformation, reverse engineering, program understanding*

1. Introduction

Often, software developers are expected to maintain poorly understood legacy systems. Unfortunately, due to the lack of proper understanding of the system, any extensions or modifications often lead to spaghetti-like code. Specifically, each modification moves the structure of the system away from its original design. Maintenance becomes increasingly difficult and if such systems are to survive, they need to be repaired or reengineered. To make maintenance easier, we need to understand the system's components and how they interact [22]. In other words, we need to extract the system's *architecture* [3,25]. Depending on what we are interested in learning about the system, we may want to create different views of the architecture (see e.g., [28]). If we determine that the *concrete* architecture of the system, which defines the way the components in the code interact, is not consistent with our mental or *conceptual* architecture of the system, then we need to investigate the possibility of repairing the system's structure[1]. We may also need to restructure the architecture to fit new operational requirements or computing platforms. In short, architectural understanding, analysis and modification are often necessary during the maintenance phase of the software-life cycle. This paper identifies *architectural transformations* that occur during maintenance (specifically during architectural understanding, analysis, and modification) and identifies the commonality between them.

Architecture extraction is subject to considerable software reengineering research; this has resulted in extraction tools such as Acacia [5], Rigi [18], PBS [20] and ManSART [28]. Given the source code, these tools determine how low-level components interact. Just as important, we need to determine the *system hierarchy* of the system: how are the modules grouped into subsystems and how are the subsystems grouped into higher level subsystems? This hierarchy or decomposition can be determined from file naming conventions, directory information, program structure information, interviewing persons familiar with the software, etc. It is our position that the component interactions (including program level dependencies such as calls from procedure to procedure), together with the system hierarchy, define the software's structure or architecture.

It is common to use a directed typed graph G to represent the system's architecture (see Figure 1): (Note that we will use Figure 1 to illustrate a number of architectural transformations.)

- Each *node* in G represents a component in the system. We can have several types of nodes. In Figure 1, we have only two types of nodes: modules and subsystems. Modules are drawn using boxes with thin lines, while subsystems are drawn using boxes with thick lines. Each node is labeled by the software component's name.

[1] We have adopted the terms concrete and conceptual architectures from Tran [26].

- Each *edge* in G represents a relation between components. We can have several types of relations. In Figure 1, we show only two types of relations: *contain* and *use*. The *contain* relation defines the system hierarchy of the software, which is a tree. There are two common ways to draw the *contain* relation; we can use nested boxes as shown in Figure 1(a), or we can use directed edges as shown in Figure 1(b). If *x* is contained in *y*, we say that *y* is *x*'s parent. We refer to nodes as *siblings* if they have the same parent and are distinct. We say that *x* is a *descendant* of *y* if *x* is nested directly or indirectly in *y* or equivalently, there is a non-empty path of *contain* edges from *y* to *x*. Besides the *contain* relation, there are dependency relations between components such as the *use* relation. In Figure 1, the *use* relation is represented as dotted edges.
- Graph nodes and/or edges may have associated *attributes*, which store information that is not conveniently expressed within the graph structure itself. Attributes may be of any type, including integer, real, text, list and table. For example, we may want to associate with each subsystem node the names of programmers who have worked on that subsystem using the *programmers_names* attribute.

In this paper we observe that once the extraction phase is complete, graph G is commonly transformed in a number of ways in order to better understand and analyze the system and to update its structure. For example, the ManSART tool recovers primitive architecture views of a software system yet these views are often too fragmented or too complex for performing software engineering work [28]. Thus, in [28], the authors proposed that such views are combined and/or simplified to produce hierarchies, hybrids, and abstractions. In general, the transformations that occur range from those simply extracting or "mining" information from G in order to gain a better understanding of the system's structure, to those actually altering G (perhaps as a part of preventive software maintenance [16]). Each of these manipulations can be thought of as applying a graph transformation function T to G to create graph G', i.e., G'=T(G). If we can collect a useful set of these transformations, this can help us understand the process of large-scale software maintenance. Furthermore, collecting, analyzing, and describing these transformations within a common framework can lead to modeling and formally specifying these transformations, which in turn can lead to their automation.

This paper takes a step towards categorizing and describing commonly used architectural transformations in the framework of graph transformations. We are concentrating on the architectural level, and so we do not include source-code transformations. In this paper, we will discuss three classes of the transformations, which are applied to the graph models of software architectures:

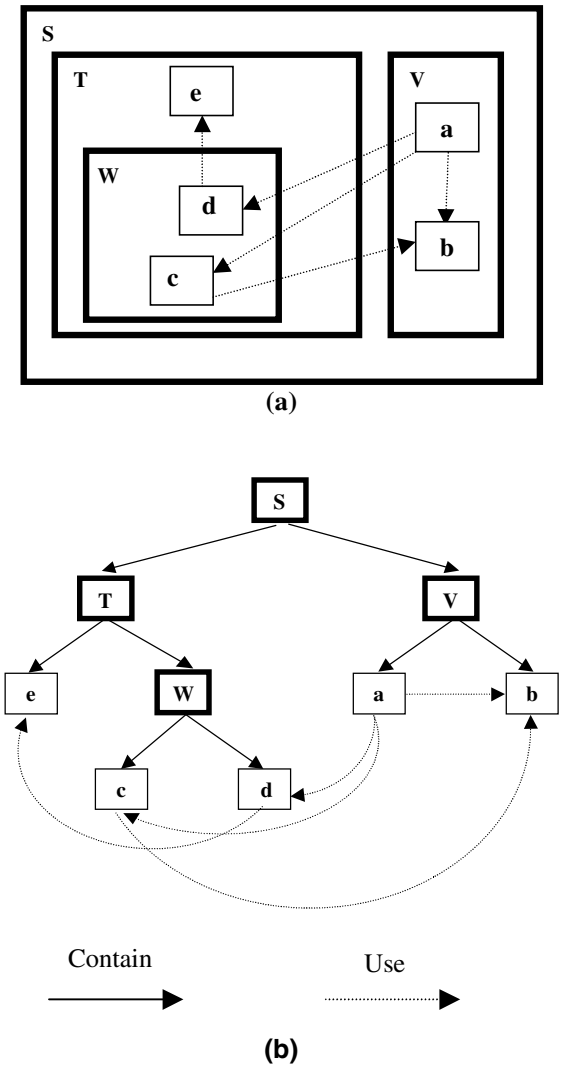


Figure 1. Two graphical representations of a software architecture. Part (a) uses nested boxes to model containment. Part (b) uses directed edges to model containment. Nodes representing subsystems have thick lines; nodes representing modules have thin lines. In this example, S contains subsystems T and V; T contains module *e* and subsystem W; subsystem W contains modules *c* and *d*; and subsystem V contains two modules *a* and *b*; module *a* uses *b*, *c* and *d*; *c* uses *b*; and *d* uses *e*.

1. *Transformations for understanding*. We use these transformations when we are building a graph model

of the system, and when we wish to explore this model to help us understand its structure. In doing this, we determine the system's hierarchical structure and we create views based on this structure.

2. *Transformations for analysis.* We use these transformations to discover various kinds of information about the software system. For example, we may want to know what modules interact in a cyclic pattern. This kind of information is commonly used to determine how we will go about modifying the system.
3. *Transformations for modification*. We use these transformations to change the system structure. For example, from our analysis we may find unexpected interactions between subsystem V and W and by moving certain modules we may eliminate these interactions.

Sections 2, 3 and 4, respectively, discuss these three classes of transformations.

2. Architecture Understanding

Tools such as RIGI and PBS extract facts from source code and use these to visualize how components such as files/modules[2] interact. For large software systems, the graph G will be huge (often containing hundreds of thousands of edges); hence directly viewing such a graph is of no help. During *architecture understanding*, we need to describe the module interactions at higher-levels of abstraction (e.g., at the top subsystem level) and also, we need to be able to simplify this information to produce various architectural views.

In the rest of this section, we will introduce the *lift* and the *hide* transformations, which help us understand a software architecture. Section 2.1 describes the lift transformation, which raises low-level relations to higher levels in the system hierarchy in order to view dependencies at various levels of abstraction. Section 2.2 discusses the hide transformation, which is used to hide the interiors/exteriors of subsystems in order to produce various views of the architecture.

2.1. Lift Transformations

It is often necessary to *lift* dependency relations to a higher level in order to study the structure at various levels of abstraction [8,12,14,19]. For example, if a function in module *a* in subsystem V calls a function in module *d* of subsystem W, then we can view that subsystem V calls subsystem W (see Figure 2). We can consider lifting[3] to be a graph transformation: applying a lifting function to a graph G adds edges to G (see Figure 2). In the rest of this section, we describe three kinds of lifting functions in terms of graph manipulations.

We begin by giving a more formal description of the lifting function applied to the architecture shown in Figure 1 with the result shown in Figure 2. If module *x* uses module *y*, and *x* is a descendant of PX and *y* is a descendant of PY, then we lift the edge (*x,y*) to (PX,PY) only if PX and PY are distinct nodes and PX is not a descendant or ancestor of PY. The resultant edges are formed between subsystem nodes. In other words, we have abstracted the module-module relations to subsystem-subsystem relations.

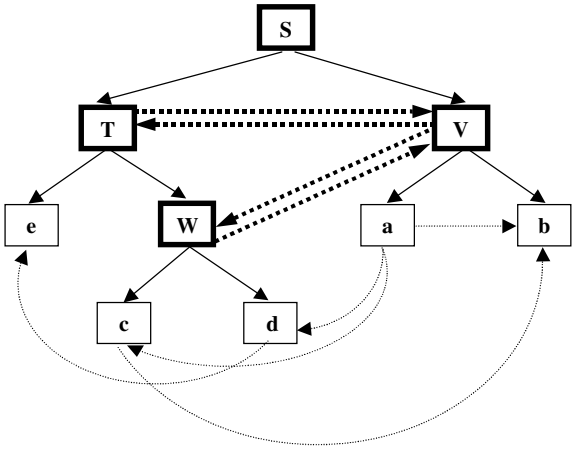


Figure 2. Lifting Transformation. Edges resulting from lifting the low-level use relations are shown as thick dashed edges. For example, since *c* uses *b*, subsystem W uses subsystem V.

Secondly, Feijs [8] defines a lifting function in terms of relation partition algebra. Here, we describe their lifting function in terms of graph transformations: for each *use* edge (*x,y*), create a new *use* edge between Parent(*x*) and Parent(*y*), only if Parent(*x*) and Parent(*y*) are distinct nodes. The new *use* edge is in turn lifted upwards in the system hierarchy one level at a time until it has reached the top level of the hierarchy. This algorithm implicitly assumes that the modules in the system are all at the same depth in the system hierarchy. This is not the case for the system shown in Figure 1 and most large industrial systems.

Lastly, Holt [12] defines a lifting function using Tarski's algebra. Holt defines a *family path* for each edge (*x,y*). If *x* is neither a descendant nor an ancestor of *y* and

[2] In this paper, we will use the terms *module* and *file* interchangeably.

[3] We will use the term *lifting* from Feijs [8]. Holt [12] refers to a *lifted* edge as an *induced dependancy*.

x and *y* are distinct nodes (as is the case for module-module *use* edges), the family path is the shortest path from *x* to *y* consisting of parents then exactly one sibling, and then children. The edges resulting from lifting (*x,y*) are all those edges that go from one node in (*x,y*)'s family path to a later node in the path. For example, when lifting the edge (*c*,*b*) of Figure 1, the edges created are {(*c*,W), (W,T), (T,V), (V,*b*), (*c*,T), (W,V), (T,*b*), (*c*,V), and (W,*b*)}. Despite the mathematical appeal of Holt's lifting function, it produces more edges than are commonly expected for a lifting function. In fact, it produces a superset of the edges produced by the first two lifting functions discussed in this section.

2.2. Hide Transformations

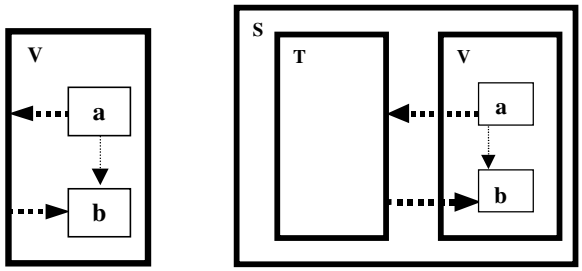
Similar to the lift transformation, the hide transformation is useful when trying to understand the structure of a software system. When a system contains several hundred files, with thousands of inter-dependencies, we need to hide parts of this information, which is not important to a particular perspective. In this section, we describe two hide transformations, hide exterior and hide interior, in terms of graph transformations.

During architecture understanding, we may want to focus our attention on one subsystem. We may want to answer questions like, which files in the subsystem are used by other subsystems? Or, which files in the subsystem use files belonging to other subsystems. When these are the questions we want to answer, we can apply the *Hide Exterior* transformation [12]. This transformation accepts the graph representing the architecture and the name of a particular subsystem we are interested in, and hides all the nodes and edges outside of the subsystem. In Figure 3a, we have applied this transformation to the graph shown in Figure 1 to hide the exterior of subsystem V. For each node *x* in V, if it is being used by something outside of V, then we added a *sell* (or *export*) edge between V and *x* since V "sells" *x* to components outside of it. If node *x* in V uses something outside of V, then we added a *buy* edge between the *x* and V since it "buys" a service outside of V. For example, in Figure 3a, V sells *b* and lets *a* buy exterior services. Finally, we deleted all edges outside of V. It would also be useful to generalize the *Hide Exterior* transformation to take as input a set of subsystems, *I*, so that only the contents of those subsystems belonging to *I*, the interactions between them, as well as the buy/sell edges to and from components that interact with anything outside of *I*, are contained in the resultant graph.

When we are not interested in the details of a particular subsystem, but rather how it interacts with the rest of the system, we can hide the interior of that subsystem using the *Hide Interior* transformation. In Figure 3b, we have hidden the interior of subsystem T as follows. For each component *x* in T that uses a component *y* outside of T, we added an edge from T to *y*. For each component *x* in T that is used by another component *y* outside of T, we added an edge from *y* to T. Finally, we deleted all components in T (i.e., nodes that are descendants of T). Like the *Hide Exterior* transformation, this transformation can also be generalized to take a set of subsystems as input.

Since hiding certain details within G has proven useful, it is also beneficial to collect relevant information from the hidden parts. For example, assume we have an attribute, `num_of_programmers`, associated with each of the nodes in G storing the number of programmers who have worked on that component. As we hide the interior of a subsystem *T* by collapsing the subtree rooted at *T* into one node, we can calculate the number of programmers who have worked on *T* given the information contained in the (hidden) descendant nodes. This is referred to as *attribute aggregation* [13].

It should be noted here that the edges formed as a result of the hide interior and hide exterior transformations can be formed using Holt's lifting transformation [12]. For example, any edge (*x,y*) produced from lifting, where *x* is a descendant of *y*, is a *buy* edge, and any edge (*x,y*), where *y* is a descendant of *x*, is a *sell* edge.



(a) Hide exterior of V (b) Hide interior of T

Figure 3. Hide Transformation shown using nested box representation. In part (a), the exterior of subsystem V of Figure 1 is hidden. Thick dashed edges are the use edges added as a result of this transformation: edge (a, V) indicates that *a* uses or "buys" some service outside of V; edge (V, b) indicates that V exports or "sells" *b* to something outside of V. In part (b), the interior of subsystem T is hidden. The thick dashed edge from T to *b* indicates that something in T uses *b*; similarly, the thick dashed edge from *a* to T means that *a* uses something in T.

In summary, we use lifting and hiding to help us understand a software system. Lifting abstracts low-level interactions into higher-level interactions. Hiding allows us to zoom in and out to concentrate on views of interest. These transformations are used in the PBS Toolkit to allow the user to navigate the structure of the software; they are specified using Tarski's algebra and calculated using Grok [12].

3. Architecture Analysis

In this section we focus on *architecture analysis*, during which we discover various kinds of information about the system that can help us restructure or modify the architecture. Questions like, "How are the concrete and conceptual architectures different and what has caused the inconsistencies?"[19] or "Which modules should be made local to other modules?"[8] or "Which modules exhibit poor information-hiding?"[15] need to be answered so that we can decide what should be changed. In this section, we describe two types of transformations that support architecture analysis: diagnostic transformations (Section 3.1) and sifting transformations (Section 3.2).

3.1. Diagnostic Transformations

Once we have extracted the concrete model of the software architecture, it often becomes evident that as the software evolved, it deviated from the intended structure or conceptual architecture [22,26]. The conceptual model may be provided by the software's architects who have determined which subsystems should interact. After lifting the low-level edges, we may determine that certain subsystems interact though they should not. For example, as shown in Figure 2, after lifting the low-level edges given in Figure 1, we determine that subsystem T uses Subsystem V and vice versa. In our conceptual model of the architecture, we may have expected that subsystem V uses T and not the other way around. In this example, we need to see what module-module edges cause the unexpected subsystem-subsystem edge (T,V). We can isolate these unexpected interactions by performing *diagnostic transformations*. We identify a high-level *use* edge between subsystems that is not expected and convert it to an *unexpected* edge. Then we *lower* [8] it (the reverse of the lifting), by identifying lower-level edges which cause the higher-level unexpected edges until we reach the bottom level (see Figure 4). Given the lifting shown in Figure 2, we determine the unexpected lower-level edges as follows. If there is an *unexpected* edge (*x,y*), then any *use* edge from *x,* or any of *x*'s descendants, to *y,* or any of *y*'s descendants, is changed to an *unexpected* edge.

The identification of inconsistencies between the concrete and conceptual model is common in reengineering software [8,19,26]. For example, Murphy [19] has developed a tool to isolate these inconsistencies, and used it to reengineer NetBSD, an implementation of Unix comprised of 250,000 lines of C code. It has also been applied to aid in the understanding and experimental reengineering of the Microsoft Excel spreadsheet product.

3.2. Sifting Transformations

During architecture analysis, we are often determining how to change the software system. This requires that we identify what parts need to be changed. In this section, we describe *sifting* transformations, which sift the software components looking for components which will play a role in the change. These transformations identify such components by examining their interrelationships with other components and update the graph by marking such components using corresponding node attributes. For example, we may wish to find and eliminate cycles in the software structure. To do so, we need to identify the components which are involved in a cycle. We can define a boolean attribute called *cycle* which is true if the component uses itself via a cycle, and false otherwise. The sifting transformation when applied to G will update G such that all components involved in a cycle will have the *cycle* attribute set to true. A more detailed example is discussed in the remaining of this section.

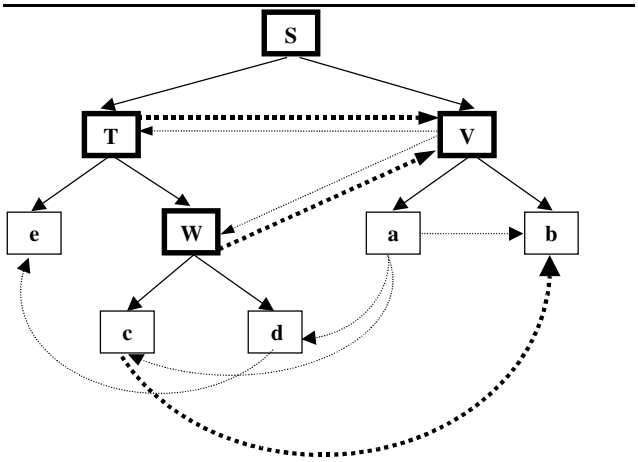


Figure 4. Diagnostic Transformations. Applying diagnostic transformations to the graph shown in Figure 2. The problematic or "Unexpected" relations are shown as thick dashed edges. Once we assert that T should not use V, then this information is *lowered* down the system hierarchy. We find out that W should not use V and *c* should not use *b*.

We may want to modify the software architecture to restructure it to fit the layering paradigm [8]. The components of the system are to be organized in layers so that each component use only components belonging to the same layer or the layer beneath it. In order to restructure the architecture in this way, we first need to identify components that are candidates for the top and bottom layers. Components which are not used but use others potentially belong to the top layer, and components which are used but do not use others potentially belong to the bottom layer. Let us define boolean attributes *top*, which is true if and only if the software component satisfies the requirements of belonging to the top layer of a layering architecture, and *bottom*, which is true if and only if the component satisfies the requirements of belonging to the bottom layer. We apply sifting transformations which inspect G and set the *top* and *bottom* values for each node. We can then use these attributes to help us restructure the architecture as a layering architecture.

In summary, we use diagnostic and sifting transformations to help us plan changes in the system structure. These create or modify edges and update nodes' attribute values, which identify problems or indicate components that may be changed or moved.

4. Architecture Modification

During the software life cycle, the need to keep the architecture up to date increases. For example, we may want the architecture to meet new requirements [4] or fit a new architectural style. Or, we may want to improve the modularity of the code by performing reclustering [23,24]. These *architecture modifications* are a part of the maintenance phase of the software life cycle. In this section, we focus on a type of modification called *repair*, which minimizes the inconsistencies between concrete and conceptual architectures. Section 4.1 describes repair transformations applied to the concrete architecture, while Section 4.2 describes those applied to the conceptual architecture.

4.1. Forward Repair Transformations

Forward repair transformations are used to minimize inconsistencies between the concrete and conceptual architectures by modifying the concrete architecture[4]. Once we have identified the unexpected relations in the concrete architecture, we apply forward repair transformations that move software components or even split components in order to help eliminate the inconsistencies. Tran [26,27] identifies two basic manipulations that he used to help minimize unexpected dependencies in the Linux and VIM architecture:

(1) **Kidnapping** moves a program entity, module or subsystem from one parent (e.g. subsystem) to a new one. For example, let us consider kidnapping component *c* from Subsystem W to Subsystem V (see Figure 1), since it doesn't use any component in subsystem W nor is it used by anything in W. If we do that, Subsystems T and W no longer use Subsystem V, and hence, we have to eliminate the unexpected edges (T,V) and (W,V) (see Figure 5). Tran [26] performs kidnapping to repair Linux's concrete architecture. For example Linux has 7 top-level subsystems, two of which are the Network Interface subsystem and the Process Scheduler subsystem [3]. The Process Scheduler subsystem unexpectedly depended on the Network Interface, and it was determined that the `inet.h` module, which is only used by modules in the Network Interface subsystem, was the cause of this dependency. By having the Network Interface subsystem kidnap `inet.h`, the unexpected dependency was eliminated.

(2) **Splitting** breaks a module or subsystem into parts. Usually, one part remains where it is, and the others are moved to other subsystems.

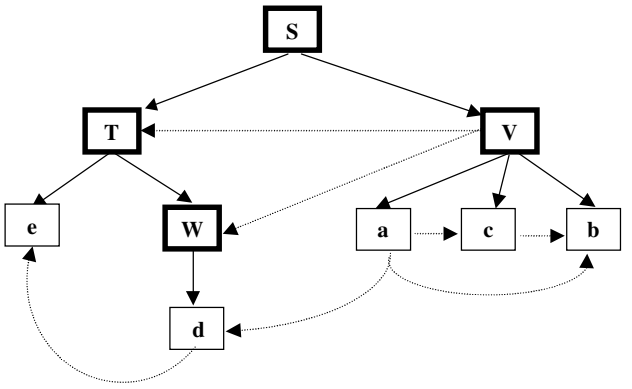


Figure 5. An example of a forward repair transformation. We have applied a kidnapping transformation to the graph shown in Figure 1. When we kidnap *c* from subsystem W to subsystem V then W and T no longer use V. Note that component *c* has all its original use edges.

For all these repair actions, we need to determine the appropriate conditions for application. In other words, when should we apply them? For example, we can say, if a component is involved in an unexpected dependency and it is not used by and does not use anything in its subsystem (like module *c* in Figure 1), then it becomes a

[4] The terms forward repair and reverse repair (see Section 4.2) are taken from Tran[26].

candidate for kidnapping. We can use a sifting transformation (Section 3.2) to determine such candidates.

When we perform a repair action on G, we need to assess whether the resultant graph G', is better or worse than the original graph G. To assess G', we can apply lifting and diagnostic transformations again to determine whether new *unexpected* edges have been created as a result of the repair. If the modified graph G' is worse, we should revert to the original graph G.

4.2. Reverse Repair Transformations

When we want to modify the conceptual architecture so that it is more consistent with the concrete architecture, we can apply *reverse repair* transformations. The main reason for wanting to modify the conceptual architecture of a system is to minimize any misunderstanding of the system so as to make maintenance easier. We will now give two examples of reverse repair.

Tran [26] performed forward repair to Linux's concrete architecture, but found that discrepancies remained between it and the conceptual architecture. Hence, he performed reverse repair on the conceptual architecture to further minimize the discrepancies. Reverse repair actions, like forward repair actions, include kidnapping and splitting as described in Section 4.1.

Another example of reverse repair transformations is discussed in Fahmy [7]. (Related work is described in Mancoridis [17].) In this case, we assume that the software architects have imposed *scoping constraints* to control how software components are allowed to interact. During software evolution, the software may no longer conform to these constraints. If this is the case, the system contains *illegal relations*. In order to eliminate illegal relations without altering the source code, we can add new allowable interactions in the conceptual model, in such a way that the illegal relations become legal. For example, Holt [11] identifies four scoping styles, one of which is the Import/Export Style. In this style, a component may export, import and use another component. X can export Y only if Y is X's child. X can import Y only if they are siblings or if X's parent imports Y's parent and Y's parent exports Y. X can use Y if they are siblings or if Y is an exported item (any number of levels of export) of X's sibling or of X's parent's imports. (This style is much like that used in various module interconnection languages (MILs) [21], as well as languages such as Java [9] and Object-Oriented Turing[10].) Without any import or export edges in the architecture shown in Figure 1, the *use* edge (*a,b*) is legal given the Import/Export scoping style; all others are not. To make the edge (*a,d*) legal, we can add *import* edges (V, T) and (V,W) and *export* edges (T,W) and (W,d).

In summary, forward repair transformations modify the concrete architecture to match the conceptual architecture, while reverse repair transformations modify the conceptual architecture to match the concrete architecture. By reconciling the conceptual and concrete architectures, we are less apt to make erroneous maintenance decisions.

5. Conclusions

This paper categorizes a number of architectural transformations that are useful during program maintenance. These include lifting and hiding transformations (Section 2), diagnostic and sifting transformations (Section 3), and repair transformations (Section 4); see Table 1. Since it is common to represent a software architecture as a typed, directed graph, we can think of these architectural transformations as graph transformations. In this paper, we have presented these in a unified way, which we hope will help us (1) *model* them so that (2) we can develop *executable specifications* for them, which (3) can lead to *tools* which automate them.

Table 1. Summary of Architectural Transformations Discussed in this Paper

Class	Type	Description
Architecture Understanding	Lifting	Lift low-level use edges up the system hierarchy
	Hide Interior/ Exterior	Eliminate information to make the structure more understandable
Architecture Analysis	Diagnostic	Given high-level unexpected edges, lower them down the system hierarchy to identify low-level unexpected edges
	Sifting	Mark components, using node attributes, that play some role in the desired change of the software structure
Architecture Modification	Forward Repair	Alter the concrete architecture to be more consistent with the conceptual architecture
	Reverse Repair	Alter the conceptual architecture to be more consistent with the concrete architecture

To model these in a common framework, we can use a graph or relation-based model. Krikhaar uses a relational

approach to model some of the described transformations [8,14], and similarly, Holt uses Tarski's algebra. Holt has been successful in using the Grok tool to execute specifications for some of these transformations [12].

Another possibility is to use *graph grammars* or *graph rewriting* [1] to model these transformations, and the PROGRES [2] tool, to execute specifications for them. PROGRES, which is an acronym for PROgrammed Graph REwriting Systems, is a visual graph-transformation language which supports the manipulation of directed attributed graphs. It shows promise; for example, Cremer [6] has used it to develop a redesign tool to migrate existing software into distributed environments. Each architectural transformation can be specified using graph rewrite rules. The application of a graph rewrite rule (1) identifies a pattern in the graph, and then (2) transforms the graph in some way based on that pattern. In other words, graph inspection as well as graph transformation is performed. Given the way we have described each of the architectural transformations discussed in this paper in terms of graphs and graph transformations, it is straightforward to specify these transformations as graph rewriting rules.

Regardless of how these transformations are specified and implemented, we hope that our framework of architectural transformations or graph transformations provides a better understanding of the maintenance of large software systems.

Acknowledgements

This work has been made possible by the first author's NSERC Postdoctoral Fellowship. The authors would like to thank Dorothea Blostein, Bob Schwanke, and the anonymous referees who provided a number of valuable suggestions, which helped improve this paper.

References

[1] D. Blostein, H. Fahmy, A. Grbavec. "Issues in the Practical Use of Graph Rewriting," *Lecture Notes in Computer Science*, Vol. 1073, 1996, pp. 38-55.

[2] D. Blostein and A. Schürr. "Computing with Graphs and Graph Transformations," Software- Practice and Experience, Vol. 29(3), pp. 197-217, 1999.

[3] I.T. Bowman, R.C. Holt, and N.V. Brewster. "Linux as a Case Study: Its Extracted Software Architecture," *Proceedings in the 21st International Conference on Software Engineering*, Los Angeles, May 1999.

[4] S.J. Carriere, S. Woods, and R. Kazman. "Software Architectural Transformation," *Proc. 1999 Working Conference on Reverse Engineering*, Oct. 1999.

[5] Y.-F.Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, Vol. 16, pp. 325-334, 1990.

[6] K. Cremer. "GraphBased Reverse Engineering and Reengineering Tools," *Proc. AGTIVE Workshop,* Aug. 1999.

[7] H. Fahmy, R.C. Holt, and S. Mancoridis. "Repairing Software Style using Graph Grammars," *Proceedings of the IBM Centre of Advanced Studies Conference*, Nov. 1997.

[8] L. Feijs, R. Krikhaar and R. Van Ommering. "A Relational Approach to Support Software Architecture Analysis," *Software-Practice and Experience*, Vol. 28(4), pp. 371-400, April 1998.

[9] J. Gosling, B. Joy, and G. Steele. The Java Language Specification, Addison-Wesley, 1997.

[10] R.C. Holt, T. West. Turing Reference Manual, 5th Edition, H.S.A. Inc., 1994.

[11] R. Holt. "Binary Relational Algebra Applied to Software Architecture," *CSRI Technical Report 345*, Computer Systems Research Institute, University of Toronto, June 1996.

[12] R.C. Holt. "Structural Manipulations of Software Architecture Using Tarski Relational Algebra," *Proceedings of the 5th Working Conference on Reverse Engineering* 1998, Honolulu, Hawaii, October 12-14, 1998.

[13] R.C. Holt. "Software Architecture Abstraction and Aggregation as Algebraic Manipulations," in *Proceedings of the IBM Centre of Advanced Studies Conference*, Nov. 1999.

[14] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. "A Two-phase Process for Software Architecture Improvement". Available at http://adam.wins.uva.nl/~x/sai/sai.html.

[15] R. Lange and R.W. Schwanke. "Software Architecture Analysis: A Case Study," *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1991, pp. 19 – 28.

[16] B. Leintz, E.B. Swanson, and G.E. Tompkins. "Characteristics of Applications Software Maintenance," *Communications in the ACM*, Vol. 21, 1978, pp. 466-471.

[17] S. Mancoridis and R.C. Holt. "Algorithms for Managing the Evolution of Software Designs," *Proceedings of the '98 International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, June '98.

[18] H. Muller, O. Mehmet, S. Tilley, J. Uhl. "A Reverse Engineering Approach to Subsystem Identification," *Software Maintenance and Practice*, Vol. 5, pp. 181-204, 1993.

[19] G.C. Murphy, D. Notkin, and K. Sullivan. "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," *Proceedings of the Third ACM Symposium on the Foundations of Software Engineering*, Oct. 1995.

[20] Portable Bookshelf (PBS) tools. Available at http://www.turing.cs.toronto.edu/pbs

[21] R. Prieto-Diaz and J.M. Neighbors. "Module Interconnection Languages," *Journal of Systems and Software*, Vol. 6, 1986, pp. 307-334.

[22] R.W. Schwanke, R.Z. Altucher, and M.A. Platoff. "Discovering, Visualizing, and Controlling Software Structure," *Proceedings of the 5th International Workshop on Software Specification and Design*, 1989, pp. 147-154.

[23] R.W. Schwanke. "An Intelligent Tool for Re-engineering Software Modularity," *Proc. of the 13th International Conference on Software Engineering*, 1991, pp. 83-92.

[24] R.W. Schanke and S.J. Hanson. "Using Neural Networks to Modularize Software," *Machine Learning*, Vol. 15, 1994, pp. 137-168.

[25] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[26] J.B. Tran and R.C. Holt. "Forward and Reverse Repair of Software Architecture," *Proceedings of the IBM Centre of Advanced Studies Conference*, Nov. 1999.

[27] J.B. Tran, M.W. Godfrey, E.H.S. Lee, and R.C. Holt. "Architecture Analysis and Repair of Open Source Software," to appear in *Proceedings of International Workshop on Program Comprehension,* 2000.

[28] A.S. Yeh, D.R. Harris, and M.P. Chase. "Manipulating Recovered Software Architecture Views," in *Proceedings of International Conference on Software Engineering*, 1997, pp. 184-194.