Wins and Losses of Algebraic Transformations of Software Architectures

H.M. Fahmy[1], R.C. Holt[1], and J.R. Cordy[2]
[1]*Dep't. of Computer Science, University of Waterloo, Waterloo, Ontario*
[2]*Dep't. of Computing & Information Science, Queen's University, Kingston, Ontario*
fahmyh@cs.utoronto.ca, holt@plg.uwaterloo.ca, cordy@cs.queensu.ca

Abstract

In order to understand, analyze and modify software, we commonly examine and manipulate its architecture. For example, we may want to examine the architecture at different levels of abstraction. We can view such manipulations as architectural transformations, and more specifically, as graph transformations. In this paper, we evaluate relational algebra as a way of specifying and automating the architectural transformations. Specifically, we examine Grok, a relational calculator that is part of the PBS toolkit. We show that relational algebra is practical in that we are able to specify many of the transformations commonly occurring during software maintenance and, using a tool like Grok, we are able to manipulate, quite efficiently, large software graphs; this is a "win". However, this approach is not well suited to express some types of transforms involving patterns of edges and nodes; this is a "loss". By means of a set of examples, the paper makes clear when the approach wins and when it loses.

Keywords: *software architecture, software maintenance, graph transformation, relational algebra*

1. Introduction

Reverse engineering plays an important role during software maintenance and reuse. It involves extracting high-level structural information from source code, which is the only accurate, complete, up-to-date representation of a program. This extracted information, such as relations between software components, is often represented as a graph; this graph is then manipulated or transformed in order to further obtain information or make the software system more easily understood. In previous work [5], we identified various transformations - specifically software architectural transformations - that commonly occur during software maintenance (see Table 1). Since the software structure is modeled as a graph, it follows that the transformations occurring on this structure can be thought of as *graph transformations* [2]. These graph transformations can be formally specified and subsequently automated. Using an executable specification language, we can test the validity of the specifications by executing them on sample structures. Recently [6], we demonstrated how to specify these transformations using a high-level, visual, graph-rewriting language called PROGRES [16]. The user enters the transformation rules, and the PROGRES tool then translates these specifications into code that is then compiled to a stand-alone prototype. Although PROGRES proved to be useful for writing specifications of architectural transformations, and in testing and debugging the specifications on small graphs, it could not handle the large graphs representing real software structures.

This paper evaluates the use of relational algebra and the Grok [9, 15] relational calculator for architectural transformations. We specify various architectural transformations using Grok and apply them to two large systems: LINUX and an I.B.M. code optimizer. The main win in using Grok is that it is efficient in processing large graphs; a loss in using Grok is that there are some transformations that are difficult to express using relational algebra. Grok has some non-relational algebraic constructs which we can use to express these transformations, yet the Grok execution times for such transformations are slow, even in processing small graphs. In this paper we characterize the wins of relational algebra, i.e., those transformations which can be easily expressed in Grok and such Grok specifications are efficient even when applied to very large graphs. We also characterize the losses, i.e., those transformations which cannot be easily expressed in Grok or whose Grok specifications are inefficient even when applied to small graphs.

This paper is organized as follows: Section 2 describes the graph representation we assume for software

architectures. Section 3 provides an overview of relational algebra, and Section 4 illustrates how we can use a tool like Grok to specify a variety of architectural transformations. Section 5 reports results in using Grok to manipulate two large software systems. Section 6 discusses the limitations of relational algebra, and Section 7 outlines how we can use Grok's non-relational algebraic constructs to work around these limitations. Section 8 concludes the paper.

Table 1. Architectural Transformations Commonly Occurring during Reengineering

Class	*Type*	*Description*
Architecture Understand-ing	Lifting	Lift low-level *Use* edges up the system hierarchy in order to study the structure at different levels of abstraction [7,9,14]
	Hide Interior/ Exterior	Eliminate information to make the structure more understandable by zooming in and out to concentrate on views of interest [9]
Architecture Analysis	Diagnostic	Given high-level unexpected edges, lower them down the system hierarchy to identify low-level unexpected edges [7,14,17]
	Sifting	Mark components that play some role in the desired change of the software structure [5, 7]
Architecture Modification	Forward Repair	Alter the extracted architecture (or concrete architecture) to be more consistent with the mental model of the software (i.e, conceptual architecture) [17]
	Reverse Repair	Alter the conceptual architecture to be more consistent with the concrete architecture [4, 17]

2. Graph Representation of Software Structures

It is common to use a directed typed graph G to represent the system's architecture (see Figure 1):

- Each *node* in G represents a component in the system. In Figure 1, we have 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 component's name.
- Each *edge* in G represents a relation between components. In Figure 1, there are three types of relations: *Contain*, *UseVar*, and *UseProc*. The *Contain* relation defines the system hierarchy, which is a tree. Figure 1 uses thick, solid edges to draw the *Contain* relation. If *x* is contained in *y*, we say that *y* is *x*'s parent. We refer to nodes as *siblings* if have the same parent and are distinct. We say that *x* is a *descendant* of *y* if 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 *UseVar* and *UseProc* relations. In Figure 1, the *UseVar* relation is represented as thin, solid edges, while the *UseProc* relation is represented as dotted edges.

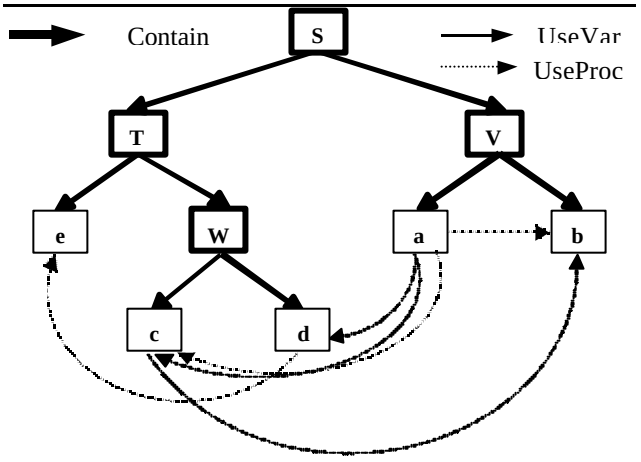

Figure 1. Graphical representation of a software architecture. Nodes representing subsystems have thick lines; nodes representing modules have thin lines. In this example, S contains subsystems T and V; T contains module e and subsystem W; subsystem W contains modules c and d; and subsystem V contains modules a and b; module a uses b, c and d; c uses b; and d uses e. There are two different types of use relations here: UseVar and UseProc.

Graph nodes and edges may have associated *attributes*, which store information that is not conveniently expressed within the graph structure itself. 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 the context of this paper, we assume the graphs are not attributed.

We now turn to how relational algebra can be used to manipulate such graphs.

3. Binary Relational Algebra and Grok

Binary relational algebra, as defined by Tarski [18] and refined by Schmidt and others, can be used to manipulate typed graphs. It provides axioms that define the effect of operators such as relational composition and relational union, along with the identity relation, the empty relation and the full (or completely connected) relation. Grok is a relational calculator which reads databases that represent sets of relations (graphs) (see Section 3.1), and which allows one to compute new relations and sets using Tarski operators (see Section 3.2).

3.1. Grok Database

In our work, we are using the PBS system to extract the "facts" from source code [15]. The format of the facts is given in ASCII and is called RSF (Rigi Standard Form) [13]. Grok reads an RSF file loading it in its memory-resident database using the "getdb" command, and then uses operators to manipulate the facts.

Figure 1 shows a graph in which the nodes are {S, T, V, W, a, b, c, d, e}. The edges are:

Contain = {(S,T), (S,V), (T,e), (T,W),
(W,c), (W,d), (V,a), (V,b)}
UseVar = {(a,d), (c,b), (a,c)}
UseProc = {(a,b), (a,c), (d,e)}

In RSF, these facts are stored as:

Contain	S	T
Contain	S	V
Contain	T	W
Contain	T	e
Contain	W	c
Contain	W	d
Contain	V	a
Contain	V	b
UseVar	a	d
UseVar	a	c
UseVar	c	b
UseProc	a	b
UseProc	a	c
UseProc	d	e

Grok also handles entity and relation attributes; specifically, it can read *TA* files [10,11], build an internal representation of it, and process it.

3.2. Grok Operators

The set of Grok operators used to manipulate a typed graph include:

Union: R1 + R2, e.g., U = UseVar + UseProc = {(a,d), (c,b), (a,b), (a,c), (d,e)}.

Intersection: R1 ^ R2, e.g., UseVar ^ UseProc = {(a,c)}.

Inverse: Inv R1, e.g., Parent = inv Contain = {(T,S), (V,S), (e,T), (W,T), (c,W), (d,W), (a,V), (b,V)}.

Subtraction: R1 - R2, e.g., U - UseVar = {(a,b), (a,c), (d,e)}.

Relational Composition: R1 o R2,which produces all edges that can be drawn by following an R1 edge and then an R2 edge, e.g., Contain o UseVar = {(V,c), (V,d), (W,b)}.

Identity: id s, the identity relation on set s. e.g., if s = {a,b,c}, then id s = {(a,a), (b,b), (c,c)}. ID is the identity relation on all entities in the database.

Transitive closure: R+, produces all edges that can be drawn by following one or more R edges, e.g., U+ = {(a,d), (c,b), (a,b), (a,c), (d,e), (a,e)}.

Reflexive transitive Closure: R*, produces all edges that can be drawn by following zero or more R edges.

Domain of a relation: dom R; e.g., dom Contain = {S, T, W, V}. DOM produces the domain of all relations in the data base.

Range of a relation: rng R; e.g., rng Contain = {T, V, W, e, d, c, b, a}. RNG produces the range of all relations in the data base.

Entities of R: ent R,e.g., ent UseVar = {a, b, c, d}. ENT produces all entities in the data base.

Set Projection: s . R produces set t: start from an entity belonging to set s and follow an R edge; the resulting entity belongs to set t. R . s produces set t: start from an entity belonging to set s and follow the inverse of R; the resulting entity belongs to set t.

Cross Product: s1 X s2, e.g., if s1 = {a,b} and s2 = {c,d}, then s1 X s2 = {(a,c), (a,d), (b,c), (b,d)}

Grok has other operators; for a complete listing refer to http://swag.uwaterloo.ca/pbs/. (The ones mentioned here are sufficient for the reader to follow through the examples discussed in this paper.) Grok is augmented with programming features (outside of the relational algebra world) such as loops and if-then-else statements. (RELVIEW [1], another system that supports the manipulation of relations and relational programming, also has such programming features.) These were added

to the language to provide more flexibility in using Grok. For example, during software maintenance, a software maintainer may want to iterate through all Module entities, determining which of them are involved in one and only one use relation. Any iteration of this kind obviously requires a loop construct.

Given a software graph like that shown in Figure 1, we can use relational algebra to define the following family relations. These relations, as well as those provided by the software graph, will be used to specify the architectural transformations in Grok. (In the rest of this paper, assume C = Contain and U = Use = UseVar + UseProc.)

Parent:	`P := inv C`
Sibling:	`S := P o C - ID`
Descendant:	`D := C+`
Reflexive Descendant:	`Do := C*`
Ancestor:	`A := P+`
Reflexive Ancestor:	`Ao:= P*`
The Set of Subsystems:	`SS := dom C`
The Set of Modules:	`MOD := ENT - SS`

In this section, we have provided an overview of Grok. We now turn to a description of a variety of architectural transformations and show how we can use relational algebra, specifically Grok, to specify them.

4. Wins: Common Software Architectural Transformations using Grok

In this section, we specify a variety of architectural transformations in Grok. For details on these transformations, the reader is referred to [5]. For each of the transformations, we assume we have extracted the software structure using some tool such as PBS or RIGI. The containment information - i.e., the system hierarchy - has been determined.

The following transformations are given by Grok scripts, which can be executed by the Grok environment. All of the scripts given below assume that we have already defined the family relations such as Parent, Sibling, and Descendant.

4.1. The Lift Transformation

The *lift* transformation is used to raise low-level *use* relations between modules to higher levels in the system hierarchy in order to study the architecture at different levels of abstraction. There are a variety of lifting functions, but a general one is as follows: if module *x* uses module *y*, and *x* is a descendant of *p* and *y* is a descendant of *q*, then we lift the edge (*x*,*y*) to (*p*,*q*) only if *p* and *q* are distinct nodes and *p* is not a descendant or ancestor of *q*. The resultant edges are formed between subsystem nodes. In other words, we have abstracted module-module relations to subsystem-subsystem relations. We can express this in relational algebra using the following statement:

```
HLU := (D o U o A) - ID - D - A
```

HLU (high-level use) is the set of new edges. We can read the above expression as, start at some node (say *p*) and follow a Descendant edge then follow a Use edge then follow an Ancestor edge reaching another node (say *q*). We add edge (*p*,*q*) to the set HLU relation provided that *p* and *q* are distinct (that is why we subtract ID from the resultant set of relations) and that there is no Descendant edge from node *p* to node *q* and that there is no Ancestor edge from node *p* to node *q*. It is important to note here that we modify the database only at the source and target nodes of the path defined by HLU above - that is, by adding relations between them.

4.2. The Hide Interior Transformation

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. For example, let us assume we want to hide the interior of subsystem T of Figure 1. For each component *x* in T that uses a component *y* outside of T, we add an edge from T to *y*. For each component *x* in T that is used by another component *y* outside of T, we add an edge from *y* to T. Finally, we delete all components in T (i.e., nodes that are descendants of T). To express this in Grok, we use six statements:

```
(1)    S := {"T"}
(2)    SD := S . D
(3)    TargetU := SD . U - SD
(4)    SourceU := U . SD - SD
(5)    NewU := (S X TargetU) +
                    (SourceU X S)
(6)    delset SD
```

In (1), we specify which subsystem whose interior we would like to hide. If we want the name to be a parameter to this Grok program, we can write, S := {$1}. In (2), we calculate the set of all nodes that are descendants of S. (We start at a node in the set S - only one node in this case - and we traverse all descendant edges; the nodes we land on belong to the set SD.) In (3), we calculate the set of nodes that are used by the descendants of S but that are not descendants of S. Similarly, in (4) we can calculate the set of nodes that use the descendants of S. In (5), we create a new set of use relations, called NewU; an edge from S to *y* is added, if *y* belongs to the set TargetU, and

an edge from *x* to S is added if *x* belongs to the set SourceU. Finally, in (6), we delete S's descendants (SD) from the database.

4.3. The Hide Exterior Transformation

Like the previous two transformations, the *hide exterior* transformation is used during architectural understanding when we are trying to simplify the information extracted from source code by creating different views. 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. This is important during architectural understanding when we want to focus our attention on one subsystem and 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? For example, if we want to hide the exterior of subsystem V of Figure 1, we would do the following. For each node *x* in V, if it is being used by something outside of V, then we add 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 add a *buy* edge between the *x* and V since it "buys" a service outside of V. Thus, V sells *b* and lets *a* buy exterior services. Finally, we delete all nodes and edges outside of V. To express this in Grok, we write:

```
(1)    S := {"V"}
(2)    SD := S . D
(3)    Ext := ENT - SD - S
(4)    SourceBuy := (U . Ext) ^ SD
(5)    TargetExport := (Ext . U) ^ SD
(6)    Buy := SourceBuy X S
(7)    Export := S X TargetExport
(8)    delset Ext
```

Statement (1) specifies S which is the subsystem whose exterior we would like to hide. Statement (2) calculates SD, the set of all nodes that are descendants of S. Statement (3) calculates Ext, the set of nodes that are exterior to S. Statement (4) calculates SourceBuy, the set of nodes that are descendants of S which use nodes in Ext. Statement (5) calculates TargetExport, those nodes in SD that are used by nodes in Ext. In other words, they are exported by S to the outside world. Statement (6) creates a new set of relations called Buy which are formed between each of the SourceBuy nodes and S. Statement (7) creates a new set of relations called Export which are formed between S and each of the TargetExport nodes. Finally in statement (8), we delete the nodes that are exterior to S from the database.

So far, we have described how we can specify three software graph transformations using Grok. The first three transformations can be used when we are trying to navigate the software in an effort to understand the software architecture. Lifting abstracts low-level interactions into higher-level interactions. Hiding allows us to zoom in and out to concentrate on views of interest.

4.4. The Diagnostic Transformation

After lifting the low-level edges (Section 4.1), we may find unexpected or undesirable interactions between nodes. For example, Tran discovered that after lifting the low-level facts extracted from a LINUX kernel, the Inter-Process Communication subsystem (IPC) unexpectedly uses the Network Interface subsystem [3, 17]. The question that arises at this point, is what are the module-module interactions causing this unexpected high-level dependency? We can isolate these unexpected interactions by performing *diagnostic transformations*. We identify a high-level *use* edge between subsystems that is not expected and mark it as an *unexpected* edge. Then we *lower* [9] it (the reverse of the lifting), by identifying lower-level edges that cause the higher-level unexpected edge. 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 marked as *unexpected* as well. In Grok, we write:

```
(1)    HLUNX :={x} X {y}
(2)    UNX := (Ao o HLUNX o Do) ^ (U + HLU)
```

UNX is the set of unexpected dependencies causing the high-level unexpected dependency, HLUNX.

4.5. Sifting Transformations

During architecture analysis, we often need to determine how to change the software system. This requires that we identify what parts need to be changed. *Sifting* transformations sift the software components looking for components that will play a role in the change. They identify such components by examining their interrelationships with other components. For example, we may wish to find and eliminate cycles in the software structure. To do so, we need to identify the components that are involved in a cycle. We may want to modify the software architecture to restructure it to fit the cycle-free layering paradigm [7]. The components of the system are to be organized in layers so that each component uses only components belonging to the same layer or the layer beneath it. In order to restructure the architecture in this way, we can first identify components that are candidates for the top and bottom layers. Components that 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. The following are some sifting transformations specified in Grok:

```
(1)    Cycle := U+ ^ ID
(2)    TopLayer := dom U - rng U
(3)    BottomLayer := rng U - dom U
(4)    UseSiblings := dom ((U o S) ^ ID)
(5)    UsedBySiblings :=dom ((S o U) ^ ID)
(6)    Outcast := (MOD - UseSiblings -
       UsedBySiblings)
```

Grok statement (1) determines those components belonging to a U-cycle. If component *x* belongs to such a cycle, then this statement creates the *Cycle* relation (*x*,*x*). Statement (2) determines those components that are candidates for the top layer of a layering architecture. Similarly, statement (3) determines those components that are bottom-layer candidates. Grok statement (4) determines those components that use at least one other component within the same subsystem. Statement (5) determines those components that are used by at least one component within the same subsystem. The last statement determines those modules that neither use nor are used by other modules within the same subsystem: they are referred to as *outcasts*. Determining such a set may prove beneficial when making decisions as to how to restructure the software; perhaps these outcasts should belong to another subsystem.

Another feature we may want to deduce to help us determine how best to restructure the software is the following. If a module *x* is used by precisely one other module (call it *y*), then such a module is a candidate for being combined with *y*. In this case, we may want to determine the relation "local_of", so if (*x*,*y*) belongs to the "local_of" relation then that means that *x* is only used by y and that we may consider moving *x* to be part of *y*. This is our first example of a "loss" using relational algebra: pure relational algebra is not sufficient in order to express this transformation. We express it in Grok as follows:

```
local_of := EMPTYREL
for x in MOD
       WhoUsesX := U . {x}
       if # WhoUsesX = 1 then
              local_of :=
                  local_of + {x} X WhoUsesX
       end if
end for
```

In this example, we make use of Grok's for-loop construct as well as its if-then-else construct, which are not inherent in pure relational algebra. Note Feijs et. al. [7] describe this type of transformation, and although they advocate the use of relational algebra in software analysis in that paper, they do not describe how this transformation can be done in pure relational algebra. This loss is exemplified in Section 5, where we will see how Grok performs this transformation much slower than other transformations. In Section 6, we will discuss other losses of relational algebra.

Having described some sifting transformations, we can now turn to an example of how we specify an architectural modification transformation.

4.6. The Kidnapping Transformation

The *kidnapping* transformation is an architectural modification transformation used to restructure the software to minimize inconsistencies between the concrete and conceptual architectures [17]. Specifically, kidnapping moves a program entity, module or subsystem from one parent (e.g. subsystem) to a new one in order to eliminate unexpected interactions (determined by diagnostic transformations). In this section, we will show how to use Grok to specify the kidnapping of a module from one subsystem to another. Specifically, we provide Grok statements that specify the moving of an entity *x* to subsystem *y*. Perhaps we have decided to move *x* since it was an outcast in its own subsystem and that it was involved in an unexpected dependency.

```
Victim := {x}
NewParent := {y}

% Delete relation between Victim and
%      its parent
old_parent := Victim . P
C := C - old_parent X Victim

% Add relation between victim and its
%      new parent
C := C + NewParent X Victim

% Now that containment information has
%      changed, recalculate the family
%      relations
P := inv C
S := P o C - ID
D := C+
Do := C*
A := P+
Ao := P*
```

4.7. Summary

In this section, we showed how Grok and relational algebra can be used to specify the graph transformations commonly occurring during software maintenance. In general these graph transformations identify some pattern in the database (graph) and modify the database in some way [1,7,12]. The next section describes the application of the Grok scripts given in this section to production software systems.

Table 2. Two Systems Studied

FEATURES	OPTIMIZER	LINUX
Number of Lines of Code	250,000	700,000
Number of Subsystem Nodes	70	129
Number of Module Nodes	941	1021
Number of Use Edges	9049	12007
Number of Contain Edges	1007	1149
Depth of System Hierarchy (i.e., containment tree)	3	6

Table 3. Grok Execution Times for various Architectural Transformations

TRANSFORMATION	OPTIMIZER	LINUX
Build family relations	1.2 sec	0.8 sec
Lifting	0.6 sec	0.7 sec
Hide Interior (of a subsystem at the highest level in the subsystem hierarchy)	0.2 sec	0.2 sec
Hide Exterior (of a subsystem at the lowest level in the subsystem hierarchy)	0.3 sec	0.3 sec
Diagnosis (given an unexpected interaction between subsystems at the highest level in the subsystem hierarchy)	0.3 sec	0.3 sec
Outcasts	5.8 sec (79 found)	2.4 sec (268 found)
Local_Of	103 sec (415 tup.)	62.7 sec (116 tup.)
Kidnapping one node	1.9 sec	1.3 sec

5. Application to Two Software Systems

In this section, we present two case studies. We applied the architectural transformations described in Section 4 to two large systems: LINUX kernel release 1.2.0 and an IBM commercial code optimizer. (See Table 2.) We used the PBS toolkit to extract the facts from both systems; the RSF files generated were then read in by Grok. The Grok scripts for the transformations were then executed on each database. Table 3 shows the timings for running the transformations on a general-purpose compute server (SUN Enterprise 450/4400 with Solaris 8 operating system).

The execution times, reported in Table 3, show the practicality of using a relational calculator, like Grok, to perform software transformations on real large software graphs. The slowest transformation was *Local_of* since this transformation requires a loop construct to iterate through all module entities. The reader should observe that although the LINUX graph is larger than the compiler graph, it took less time to apply the transformations to the LINUX graph than to the compiler graph. Perhaps this is due to the fact that the LINUX system's structure is more hierarchical.

Although these results are encouraging, Grok and other relational algebraic techniques are limited to which patterns they can identify in the graph. The next section discusses such losses of using relational algebra.

6. Losses: Limitations of Relational Algebra

We have seen in the previous section that Grok is able to process large software systems reasonably well. In Section 4.5, we discussed how some transformations require that we step through the entities to ascertain certain properties. This requires a loop construct that is not inherent of relational algebra, and Grok has it specifically to be able to handle such types of transformations. There are other limitations to relational algebra. This section describes some of them.

There are two main losses of using relational algebra. First of all, we are restricted in the types of patterns we can find using algebraic expressions. This section gives an example of such a pattern. Second of all, once we have determined that the pattern exists in the database, there is no inherent way of marking the whole matched pattern. More specifically, there is no inherent way to mark and bind each of the nodes and edges visited. For each traversed path, all we have is a relation (s,t), where s is the source node of the path and t is the target node; what nodes/edges visited along the path from s to t are lost. Thus relational algebra is not well suited for transformations where we need to remember or access any of the nodes or edges along the visited paths. We now discuss an example that illustrates these losses.

In redesigning a software's architecture, software maintainers may want to merge tightly coupled modules into a single module. In Figure 2, the transformation

shown illustrates that if we have four modules which interact with each other completely (i.e., each module interacts with the other three), we can merge them into one new module.

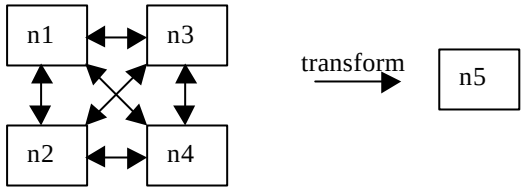

Figure 2. The Merge Transformation. This transformation states that if we have four modules represented by n1, n2, n3, and n4 which interact with each other completely, we can merge them into one new module. Note that a two-way arrow shown indicates that there is a two-way Use relation.

To perform the "merge" transformation, we need to first identify the pattern matching the left-hand side of the above rule. We can try to represent this pattern as a path starting from n1 and returning to n1, where some nodes and edges are visited more than once: (There are obviously many ways to do this; however there is no way to do this without having repeated nodes or edges.)

```
P :=  id(n1) o Use o id(n2) o Use o id(n1)
      o Use o id(n3) o Use o id (n1) o Use
      o id(n4) o Use o id(n1) o Use o
      id(n2) o Use o id(n4) o Use o id(n2)
      o Use o id(n3) o Use id(n2) o Use o
      id(n4) o Use id(n3) o Use o id(n4) o
      Use o id(n1) ^ ID[1]
```

This expression for P attempts to trace out a path that corresponds to the edges of the left-hand side of the merge transformation. Unfortunately, it fails to capture the meaning of the pattern, because terms such as n2 do not correspond to particular nodes. Since pure relational algebra does not allow us to mark the nodes along the path and hence, bind them to variable names, then we cannot perform *unification*, whereby variables with the same name are bound to the same node. It is important to note here that even if we had the ability to unify variables, P simply represents a set of identity relations at the nodes matching n1; it does not represent the set of whole matched patterns. Thus, we cannot conveniently complete this transformation by replacing the interacting modules by one module since we do not know, at the very least, where the nodes matching n2, n3, and n4 are.

[1] We need to intersect the expression with the IDENTITY relation in order to ensure that we end off where we started from.

In summary, relational algebra is not well suited for transformations which (1) involve patterns that require unification, and/or (2) require knowing where the instances of the pattern are. Next, we outline how we make use of Grok's non-relational algebraic features to work around these limitations.

7. Generalized Pattern Matching Using Grok

In this section, we describe an approach that we developed in order to use Grok to perform generalized pattern matching. Essentially, the approach finds all subgraphs in the database which are isomorphic to the pattern we are searching for. Because subgraph isomorphism is an NP-complete problem, the algorithm is inherently slow.

The main idea behind the algorithm is to generate combinations of nodes in the database that match the pattern. If the pattern contains four nodes, n1, n2, n3 and n4, and the graph contains n nodes, then the number of combinations is $O(n^4)$. Obviously, we want to reduce this number; thus we implemented a *discrete relaxation* [8] algorithm to reduce the *node sets* assigned to n1, n2, n3 and n4, where each node set represents the set of possibilities for that particular node. We will not get into the details of this component of the algorithm as it is beyond the scope of this paper.

Once we've reduced the node sets as much as possible, we then generate each combination and we test it against the relations given by the pattern. Once we have a match, we need to save this information and somehow mark the entities. To do so, we can use a two-dimensional array P[I,N], where I is the number of instances found of the pattern, and N is the number of nodes in the pattern. (For example, P[2,4] records the 4^{th} node of the second instance of the pattern.) Grok does not have arrays, but we simulated arrays using Grok's ability to add prefixes and suffixes to elements of a set. We use these features to create a set, P, whose entities have prefixes and suffixes indicating which instance they belong to and which node in the pattern they are bound to.

Let us consider a simple example in order to make this approach clear. Figure 3 shows the lifting transformation (Section 4.1) rule represented graphically: Since we showed in Section 4.1 how this transformation can be specified using basic relational algebraic operators, we do not need to use our more general algorithm to find the matches. Nonetheless, this transformation is a good example to show how working around relational algebra's limitations, we can tremendously slow down Grok's performance.

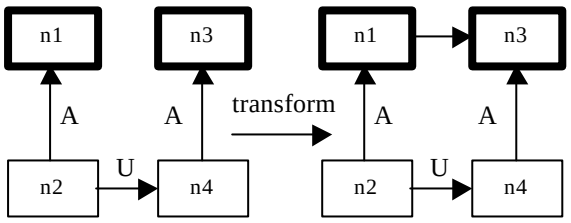

Figure 3. The Lifting Transformation. A = Ancestor relation; U = Use relation; HLU = high-level use relation; Nodes representing subsystems are drawn using boxes with thick lines; Nodes representing modules are drawn using boxes with thin lines.

In finding all the subgraphs in the graph shown in Figure 1 matching the left-hand side of the rule, we first assign nodes n1 and n3 to all subsystem nodes in the database, and assign nodes n2 and n4 to all module nodes. After using discrete relaxation to reduce the node sets further, we test each combination against the relations given by the pattern; if it matches, we add the combination to P, a simulated two-dimensional array. The Grok script used to perform lifting in this way took 23.6 seconds to produce P:

$$P = \begin{bmatrix} T & c & V & b \\ V & a & T & c \\ V & a & T & d \\ V & a & W & c \\ V & a & W & d \\ W & c & V & b \end{bmatrix}$$

Each row represents one match. The first row indicates that node T of Figure 1 has been bound to node n1 of the pattern, node c has been bound to node n2 of the pattern, node V has been bound to node n3, and lastly, node b has been bound to node n4 of the pattern. The reader should note that an entity can belong to more than one match.

Recall that encoding the pattern simply as a path as given in Section 4.1 and not in this generic way, Grok was able to perform lifting on a much larger graph in less than 1 second (Table 3). Also, PROGRES, a general graph-matching tool, is able to perform lifting on the small graphs such as that shown in Figure 1 in less than 1 second, but takes approximately 10 minutes to process the code optimizer or LINUX.

In summary, in this section, we described how we can work around the limitations of relational algebra using Grok. Since Grok is not tuned for the combinatorics involved in performing subgraph-isomorphism testing, the algorithm is slow even for small graphs.

8. Conclusions

Recently, there has been considerable work in software maintenance involving the use of relational algebra to understand, analyze, and modify software architectures [7, 9-12]. This paper provided an overview of the types of transformations we can perform using relational algebra. We focussed on one tool - Grok - and using this tool, we illustrated the wins and losses of the algebraic approach in identifying and manipulating patterns in software graphs. In summary, many of the types of transformations that commonly occur during software maintenance can be specified easily and elegantly in Grok. These executable specifications can be used to process large software graphs relatively efficiently. These are wins. However, there are losses as well. Relational algebra is not well suited for generalized pattern matching; the types of transformations that cannot be easily specified using relational algebra are quite simply those that require storing some or all of the nodes and edges which are visited along the path that represents the pattern. We showed in this paper how we use some of Grok's features to work around this limitation, yet even for small graphs, Grok was slow. In the future, we hope to explore ways to make Grok more efficient in handling general pattern matching.

References

[1] R. Behnke, R. Gerghammer, T. Hoffmann, G. Leonik, and P. Schneider. "Applications of the RELVIEW System," in Berghammer R., Lakhnech Y. (eds), *Tool Support for System Specification, Development and Verification, Advances in Computing Science*, Springer-Wien, pp. 33-47, 1999.

[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] H. Fahmy, R.C. Holt, and S. Mancoridis. "Repairing Software Style Using Graph Grammars," *IBM Proceedings of the Seventh Centre for Advanced Studies Conference (CASCON '97)*, Toronto, Canada, Nov. 1997.

[5] H. Fahmy and R.C. Holt. "Software Architecture Transformations," *Proceedings of the International Conference on Software Maintenance*, San Jose, Oct. 2000.

[6] H. Fahmy and R.C. Holt. "Using Graph Rewriting to Specify Software Architectural Transformations," *Proceedings of Automated Software Engineering*, Grenoble, France, Sept. 2000.

[7] 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.

[8] T. Henderson, Discrete Relaxation Techniques, Oxford University Press, New York, 1990.

[9] 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.

[10] R.C. Holt. "Software Architecture Abstraction and Aggregation as Algebraic Manipulations," *Proceedings of CASCON '99*, Toronto, Canada, Nov. 1999.

[11] R.C. Holt. "An Introduction to TA: The Tuple-Attribute Language," Available at http://plg.uwaterloo.ca/~holt/papers/ta.html, Nov. 1998.

[12] C. Lange, H. Sneed, and A. Winter, . "Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools," *Proceedings of the 9th International Workshop on Program Comprehension*, Toronto, Canada, May 2001.

[13] 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.

[14] 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.

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

[16] PROGRES. Available at http://www-i3.informatik.rwth-aachen.de/research/progres/release.html.

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

[18] A. Tarski. "On the Calculus of Relations," *Journal of Symbolic Logic*, Vol. 6, No. 3, 1941, pp. 73-89.