Structural Manipulations of Software Architecture Using Tarski Relational Algebra

Richard C. Holt

University of Waterloo, Canada, holt@uwaterloo.ca, http://plg.uwaterloo.ca/~holt

Abstract

A software architecture is typically drawn as a nested set of box and arrow diagrams. The boxes represent components of the software system and the edges represent interactions. These diagrams correspond to typed graphs, in which there are a number of "types" or "colors" of edges, and in which there is a distinguished "contain" relation that represents the system hierarchy (the nesting of boxes).

During reverse engineering, one often transforms such diagrams in various ways to make them easier to understand. These transformations include edge aggregation, box abstraction (closing a box to hide its contents), and box separation (separating a box from its surrounding system). Such transformations are essential in helping make software architecture diagrams useful in practice.

This paper shows how structural manipulations such as these can be specified and automatically carried out in a notation based on Tarski's relational algebra. The operators in this algebra include relational composition, union, subtraction, etc. These operators are supported in a language called Grok. Grok scripts have been used in manipulating the graphs for large scale software systems, such as Linux, to help in program visualization and understanding.

1. Introduction

It is common when describing the architecture of a software system to draw it as a box-and-arrow diagram [13]. When the architecture directly shows the structure of the implemented system (this is called the *concrete* or *as-built* architecture [11]), the boxes in the diagram often represent functions, files, or subsystems that contain files as well as other subsystems (see Figure 1 for an example). This containment gives the hierarchic decomposition of the system, which is in most cases a tree or forest. In this paper we show how, when these diagrams are represented as typed graphs (with more than one kind of edge), they can be algebraically manipulated to create new graphs (views of the original graph) which correspond to diagrams that people commonly draw. The algebraic operations, such as relational union and relational composition, have been incorporated into an executable language called Grok, which allows us to specify and re-use common architectural transformations. With this approach we are able to automate certain steps in the process of reverse engineering.

Views: Moving From Complexity to Comprehensibility. When one attempts to view the relations that appear in the source code for a large software system, one too often draws a diagram with hundreds or thousands of boxes and correspondingly more edges. For example, Carmichael's paper [5] gives such a diagram that shows the raw structure of a commercial Cobol compiler (front end only), written in the C language and consisting of roughly 300,000 lines of code. In his diagram, each of the 300 boxes represent files, and details at the level of functions and variables have been hidden. The arrows represent *includes* from one file to another. Although much detail has been omitted, the graph remains too complex to provide us with much meaningful information. We refer to such pictures as *Slonim diagrams* because Jacob Slonim objected to being shown any more of them on the grounds that they are a waste of time. One challenge in reverse engineering is to transform Slonim diagrams into comprehensible and useful views of the system.

A Motivating Example. Figure 1 shows a diagram, transformed from an earlier Slonim diagram, of a large (250,000 lines of code) system. This is a screen dump from the PBS (Portable Bookshelf) system [8,9]. In this case the system is a commercial compiler back end (one of its front ends is the Cobol compiler just discussed), which generates machine language from intermediate code representing the user's program.

We will briefly discuss the main subsystems shown here to illustrate how we use such a diagram to help us understand the system. The top left box (subsystem) is the called the *Optimizer*; it takes a user's program, as represented in an abstract machine language, and optimizes it by successively running sub-optimizations, which are carried about by boxes (subsystems) within the Optimizer. The top middle box is the *Allocator*, which takes the optimized but abstract code and adapts it to a particular target computer, such as an RS6000 microprocessor. It does this using a number of subsystems inside the Allocator box.

The top right box is the *Assembler*, which takes the optimized code from the Allocator and writes it to a file. The bottom subsystem, called *Services*, contains supporting routines as well as global data structures used by the entire back end. The smallest boxes in Figure 1 are subsystems, typically containing 10 to 25 source files.

In this diagram, as actually displayed on a computer screen, the user can read the names of each subsystem in each box and can access the diagram and documentation for each subsystem [8] by clicking its box. We have hidden all edges except "calls" edges coming out of the driver module in the Allocator, in order to avoid cluttering the picture. The PBS system allows the user to selectively "turn on" arrows to/from any box, thereby allowing a systematic exploration of interactions of the systems parts (other viewers such as Rigi [12] provide analogous facilities).

Figure 2 shows details of one particular subsystem, the DS (Dead Store Removal) subsystem, from Figure 1. This subsystem performs the optimization of removing "store" instructions whose stored values are never used. This c*lient-server* view, which extracted from Figure 1, shows only the DS subsystem and its clients (boxes which use DS) and its suppliers (boxes which are used by DS). There are two types of arrows in this view, red ones representing function calls and blue ones representing data references (these colors are not visible in the black and white diagram).

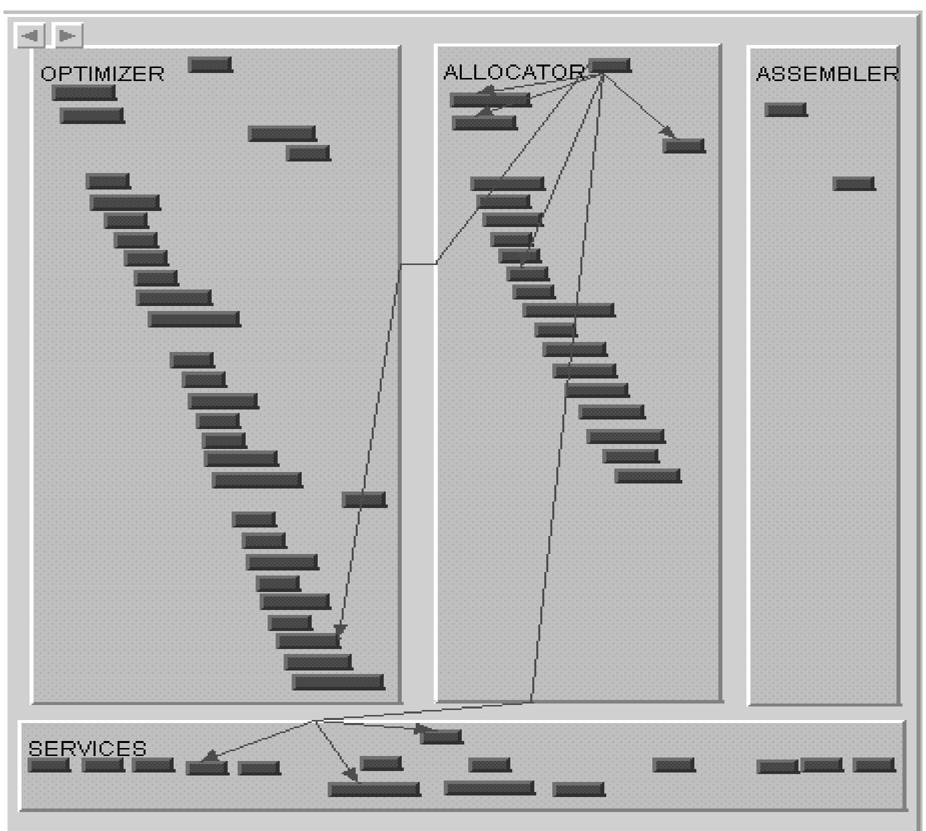


Figure 1. Top view of concrete software architecture of 250,000 line software product

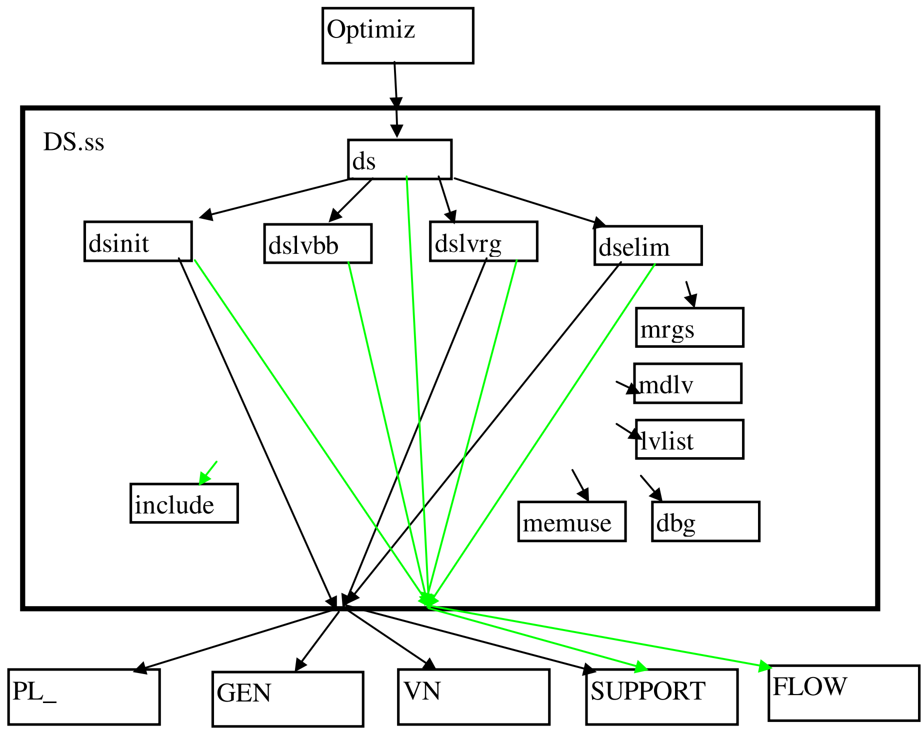


Figure 2. Client-server view of one subsystem of the 250,000 line product

It often happens that there is a clutter of arrows within a diagram. There are various methods for dealing with this clutter. In Figure 2, for example, some but not all of these arrows are shown. The bottom left box in DS is an "include file" that contains data used throughout DS. Since every other DS box references this box, arrows to it are omitted. Similarly boxes on the right are routines that are used throughout DS, so the arrows to them are omitted.

The manipulations that produce diagrams such as Figures 1 and 2 are the subject of this paper.

Creating Architectural Views. Figures 1 and 2 were created using the process illustrated in Figure 3. This process uses the PBS system [9] to do reverse engineering; similar processes have been described for other systems [4,12,3,11]. As shown in Figure 3, the source code is parsed to extract "facts", such as what functions call what functions and what variables are referenced by what functions. These facts are manipulated to determine new facts and to eliminate certain facts. The resultant facts, which are effectively edges in a graph, are fed to a *Layouter* program that generates (x,y) locations as attributes for boxes to determine how to draw a corresponding diagram. Finally, the diagram, along with its related attributes is presented to the user by means of a browser (see Figure 1 for a screen dump from the PBS browser).

As Figure 3 shows, the hierarchic decomposition (tree organization) of the system is fed into this process in addition to the facts derived from the source code. While there exist reasonably good parsers to extract facts from a software system's source code, it remains an unsolved problem to automatically produce a hierarchical clustering from the source code that closely matches the mental model of the systems' designers. The pragmatic approach taken to this clustering problem [15] is to determine this hierarchic structure by interviewing people familiar with the system, examining any available documentation, and making use of any naming conventions in the source code. (Of course this approach may fail in legacy systems in which the designers and the documentation are not available.) The hierarchic decomposition can be represented as more "facts", e.g., the fact that the Optimizer subsystem contains the DS subsystem, which are added to the facts derived from the source code.

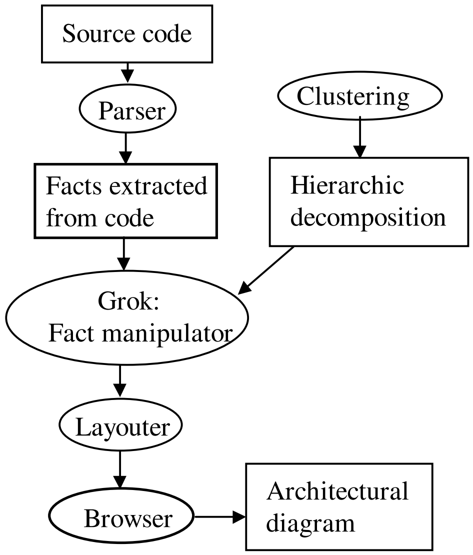


Figure 3. Process of view creation

Overview of Paper. The next section of this paper describes a formalism, Tarski's binary relational algebra, which we use as a basis for structural transformations such as those used in creating Figures 1 and 2. Then the following section describes the *hide* transformation, which eliminates information about certain parts of a system's graph to allow the user to concentrate on other parts. The *hide* transformation is used to illustrate how various transformations can be defined by operators of binary relational algebra. This is followed by a section that shows how the Grok language, which implements these relational operators, is used to automatically carry out the transformations specified in terms of algebraic operations.

2. Graphs with Algebraic Operators

This section defines typed graphs, introduces algebraic operators on such graphs, and explains how these graphs can be stored as ASCII files.

Typed Graphs. We will concentrate on graphs in which there are a fixed number of *types* (sometimes called *colors*) of edges. Figure 4 shows such a graph in which the nodes are r, a, b, v, w, x, y and z. The relations (edge types) are:

- C = { (r,a), (r,b), (a,v), (a,w) (a,x), (b,y), (b,z) }
- I = { (a,b) }
- E = { (b,y) }
- U = { (v,w), (x,y) }

The relation names in this example have been chosen to be suggestive as follows: C for *Contain*, I for *Import*, E for *Export*, and U for *Use*. The two diagrams in Figure 4 show the same graph, drawn using different conventions. In the lower diagram, Contain is shown explicitly using edges, while in the upper diagram, Contain is shown implicitly using nested boxes. It is common in architectural diagrams to show containment by nesting, as is done in Figures 1 and 2.

Algebraic Operators. Tarski's [14] classical paper on the calculus (or algebra) of binary relations gives an axiomatic specification of a set of operators for manipulating typed graphs. This set of operators, to which we have added transitive closure, include:

- *Union*: R1 + R2, e.g.,. I + E = {(a,b), (b,y)}
- *Intersection*: R1 ^ R2, e.g., E ^ C = {(b,y)}
- *Subtraction*: R1 - R2, e.g., C - E = {(r,a), (r,b), (a,v), (a,w) (a,x), (b,z)}
- *Inverse*: inv R1, e.g., inv E = {(y,b)}
- *Relational composition:* R1 * R3, which produces all edges that can be drawn by following an R1 edge and then an R2 edge, e.g., I * E = { (a,y) }
- *Identity:* id, the set of all edges that loop directly back to nodes, e.g., { (r,r), (a,a), (b,b), (w,w), … }
- *Transitive closure:* R+, e.g., C+ is equal to the union of C and { (r,v), (r,w), (r,x), (r,y), (r,z) }
- *Reflexive transitive closure*, R*, which is the union of R+ and *id*

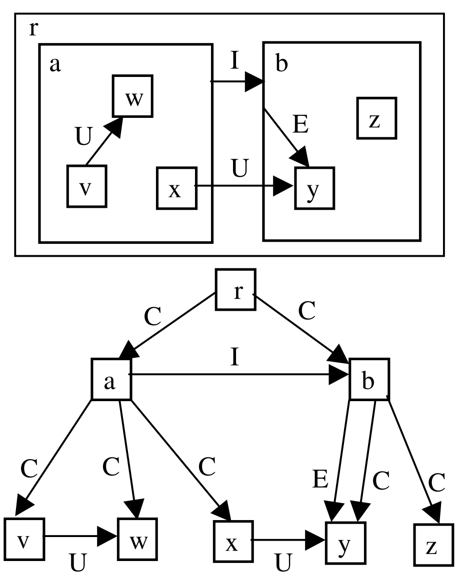


Figure 4. Example typed graph

In this paper we are using ASCII characters as operators, such as + for union, that are directly accepted by the Grok processor, rather than classical mathematical notation, such as $\cup$ for union.

In a later section of this paper, we will show how this set of operations on typed graphs is used to transform a messy set of facts, such as those in a Slonim diagram, into a more tractable view.

External Representation of Graphs. In our applied work, we represent the edges of a typed graph in external files using a simple ASCII format called RSF (Rigi Standard Form [12]). For example, this RSF file:

```
call P Q
include F G
store P X
contain Parser P
```

contains four edges (facts). The first is an edge from node P to Q with type *call*, which we can interpret as a call from function P to function Q. As well, we can interpret that file F includes file G, procedure P stores variable X and the Parser subsystem contains procedure P.

Having introduced Tarski binary relational algebra, we will next describe a key example of a graph transformation, and after that we will show how algebraic operators are used to specify and carry out these transformations.

3. The *hide* Transformation

To make a Slonim diagram comprehensible we need to eliminate some information from it. Surprisingly, a good way to start is first to add more information, namely, the system's hierarchic decomposition (Figure 3 shows the adding this information). Using this information, which corresponds to the C relation in Figure 4, we can simplify the diagram by hiding certain parts of it. For example, in Figure 1, all contents of the small boxes are hidden, leaving us with a clear view of the top level hierarchic structure which does not let us see all the way down to the file or function level. In Figure 2, we have shown the DS subsystem but have hidden the surrounding system (except for DS's clients and suppliers).

We will explore the "hide" transformation in considerable detail. We will show how it can be formalized and how its formalization can be directly executed. This formalization provides several advantages. One is to clarify the exact nature of this hiding. Another is to make it easier to communicate from one worker to another (or from a teacher to a student) our notion of hiding. Other advantages are discussed later in this paper.

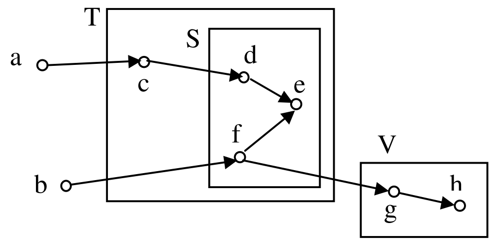
Figure 5A shows an example graph called G. In G, we might want to hide the interior of part S to simplify the diagram. To do this we can blacken out S's interior, or equivalently, shrink S's oval to a dot. When doing this, we will take any arrow entering S, such as (c,d) and shorten it to attach to S, to become (c,S). Similarly, any arrow leaving S, such as (f,g), is shortened so it starts at S and goes to g, so it becomes (S,g). Given graph G, the new graph that hides the interior of S is denoted as *hide* (G,S).

As a further example, consider hiding the interior of both T and V, as shown in graph H in Figure 5B. The diagram for H is similar to Figure 1 in which the interiors of all low level subsystems were hidden. The transformations to hide T and V are the same as that for hiding S; in particular, arrows leaving T or V are shortened to locate T or V, respectively. We can denote graph H as either hide(hide (G,T),V) or hide(hide(G,V),T).

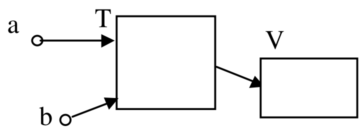
To summarize, our intuitive definition of hiding is: *The **hide** transformation takes a graph G that includes a hierarchical (tree) containment relation C and another relation R and a particular node S, and eliminates all descendent nodes of S while replacing each edge from outside S to inside S by an edge to S, and each edge from inside S to outside S by an edge from S.* If there are several relations, R1, R2, etc., edges of each type are shortened as described for R.

When should we use *hide*? We will now explain that *hide* is a useful transformation when R (the relation between nodes) has transitive and "optimistic" properties.

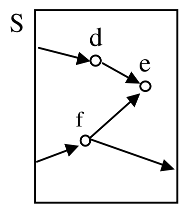
It is interesting that the *hide* transformation is so "natural" in certain situations --- if you ask someone to eliminate the architectural details about T and V in graph G, there is a good chance that he or she will draw the same diagram as that in Figure 5B. However, sometimes *hide* is not so natural. For example, if relation R represents quite a different kind of concept, such as the "disjoint from" relation, *hide* is not longer useful. To see when it seems natural, we will take a closer look at what happens when a node is hidden.



Part 5A: Graph G



Part 5B: Graph H = hide(hide((G, T),V)



Part 5C: Graph I = hideExt (G, S)

Figure 5. Hide transformations.

There is something surprising about hiding node S in G, namely, in graph hide(G,S) you can follow directed edges from c to S to g, but in graph G without this hiding, you cannot follow directed edges that lead you from c to g. Stating this mathematically, the transitive closure of R in hide(G,S) includes edge (c,g), but this edge does not appear in the transitive closure of R in the original graph. Why has *hide* changed R in this way? The explanation is that *hide* is an information-losing transformation; it eliminates information about the part of the system inside S. When the *hide* transformation shortens arrows into/out of S, it in effect "optimistically" assumes that every node in S is (or might be) connected to every other node in S. This "optimism" is the cause of the extra connectivity that we observe when there is hiding. The result is that *hide* maintains and sometimes enhances the (transitive) connectivity of nodes. This way of modelling hidden edges provides a good (natural) fit for architectural relations such as "static dependency" and "data flow" which are common in architectural diagrams.

Eliminating the Exterior of a Part. While it is common to hide the interior of parts of a system, it is also common to hide a part's exterior, as is done in Figure 2. Similarly, in figure 5C, we have eliminated the exterior of S. Each edge starting in S leading outside, such as (f,g), is shortened to end at S (at S's periphery) to become (f,S) and each edge outside of S leading inside, such as (c,d), is shortened to start at S to become (S,d). We will call this transformation "hide exterior", or *hideExt*, so the graph in 5C is hideExt(G,S). It is essentially the same transformation as *hide* proper if we consider that what is being hidden are the nodes outside of S instead of inside S, with the complementary interpretation of edge shortening.

Figure 2 shows the client-server view of the DS subsystem in which nodes exterior to DS are eliminated except for those nodes with edges to/from nodes in DS. This client-server view is a variation of the *hideExt* transformation. There is also a variant of *hide* proper in which those interior nodes within a hidden node that have direct connections to the outside are retained; this hide variant is useful because it allows us to see how a surrounding system directly interacts with the interior parts of a hidden part without the complexity of seeing the entire interior of the hidden part.

Edges Induced by Hiding. When there is a hide transformation, a particular edge, such as edge (f,g) in graph G in Figure 5A, may be shortened to *induce* another edge. For example, when both T and V are hidden, edge (f,g) induces edge (T,V). When the exterior of S is hidden, edge (c,d) induces edge (S,d).

We now ask the question*: For a given edge E, what edges can it induce, given any sequence of interior or exterior hiding?* To illustrate, suppose that E is edge (f,g). For such an edge, we define its *family path* as the sequence consisting of parents (containers), then (at most one) sibling and then children (containees) that we pass through to get from f to g. (Recall that the containment relation is a tree.) By inspection of Figure 5A, we can see that the family path for (f,g) is (f, S, T, V, g). In other words, to get from f to g following parent/sibling/child links, we go to f's parent S then S's parent T then T's sibling V and finally V's child g In general, the family path for nodes x and y is the shortest path from x to y following a number of parent edges, at most one sibling edge, and then a number of child edges.

Now to answer our question: *The edges that can be induced by edge E are all those edges that go from one node in E's family path to a later node in the path.* In the case of E = (f,g), the set of induced edges consists of single hops (f,S), (S,T), (T,V), and (V,g); double hops (f,T), (S,V), and (T,g); and triple hops (f,V) and (S,g), all along the family path. For completeness, it also contains a quadruple hop (f,g) which is the original edge. In Figure 5A, you can see that these edges are all the subsegments of the line drawn from node f to node g. It is easy to show that E has order n^2 induced edges where n is the height of the containment tree (really, the distance from f or g up to the sibling level).

An induced edge that crosses from one part to another, for example, edge (T,V), is sometimes called an *Import* edge, because it represents a high level dependency between disjoint components. An induced edge that descends from a part of the system to one of its internal nodes, e.g., (S,d), is called an *Export* edge, because it indicates that a part of S is used by other parts of the system. (Exporting a part is similar to making it *public* in C++.) An induced edge that ascends from a part to one of its ancestor nodes is called a *Buy* edge, e.g., (f,S), because it tells us that a part of S uses (buys) services that are outside of S. (See Fahmy [7] for a discussion of Import, Export and Buy relations.)

A further question we ask is: Why, other than intellectual curiosity, would we want to determine all of the induced edges? One reason is that it shows us all the interactions that low level edges cause among high level entities. Another reason is to allow us to precompute all the edges that could need to be shown as a person browses through an architectural diagram, hiding interiors and exteriors of various parts of the system. If the diagram is drawn such that all edges induced from a given edge are drawn on top of the inducing edge, then hiding and later exposing parts of the system can be shown on the screen by always drawing all induced edges and blackening out hidden parts.

The next section explains how edge induction can be specified formally in terms of algebraic operators, and the section after that explains how these specifications can be executed as statements in the Grok language.

4. Specification of Edge Induction

This section shows how edge induction due to hiding can be defined in terms of binary relational operators; it is rather technical and could be skipped on first reading.

Family Relations. First we need to define certain "family" relations: parent, descendent, ancestor and cousin. Given a contain or child relation *C*, the *parent* relation is defined as *P* = inv C (C run backwards). The (proper) *sibling* relation *S* is (P * C) - id, that is, non-identical children of the same parent. The (proper) *descendent* relation is *D* = C+ (transitive closure of containment) and the (inclusive) *descendent* relation is *Do* = C* (same as D but includes identity). Similarly, the proper and inclusive *ancestor* relations are *A* = P+ and *Ao* = P*. Of course, A = inv D and Ao = inv Do. We define a *super cousin K* relation as follows. Every K edge is an edge determined by following any number of parent P edges then a sibling edge S and then any number of child C edges: K = P* * S * C*, where postfix * is transitive closure and infix * is relational composition. For convenience, we will refer to super cousins as simply *cousins*. It follows that for a given containment tree, defined by C, the set of all edges that can be drawn between the nodes in the tree is partitioned by A, D, id and K; in other words, every edge is an ancestor, descendent, identity or (super) cousin edge, exclusively. Note that the definitions of P, A, Ao, D, Do, and K have all been defined strictly in terms of C and binary relational operators, namely the operators inverse (inv), transitive closure (postfix + and *), subtraction (-) and composition (infix *).

We are now ready to give formulae for edge induction for a given relation, which we will call U (for *Use*) (instead of R) that occurs in decomposition tree C. We will simplify our presentation by assuming that U has no self loops (it is disjoint for id), and we will partition U into Uk (cousin edges), Ud (descendent edges) and Ua (ancestor edges). The set of edges induced by all possible hiding (interior and exterior) by edges in U will be called U'. We partition U' into Uk' (cousin edges), Ud' (descendent edges) and Ua' (ancestor edges). (U' will have no id edges.) Note that cousin/descendent/ascendent edges correspond, respectively, to Import/Export/Buy edges.

Induced Cousins. We can now give the formula for induced cousin edges Uk'; this is simply a formalization of our intuitive description from the previous section of this paper. The formula states that there is an induced edge (x,y) in Uk' whenever x's descendent t uses (via Uk) node u which has y as an ancestor, with the caveat that x and y must be cousins:

Uk' = (Do * Uk * Ao) ^ K

We will now explicate the formula in some detail. It states that an induced edge that is a cousin (Uk') is any edge that can be created by following any number of C edges (Do), then an explicit cousin use edge Uk, and then any number of parent edges (Ao), but restricting the result to be a cousin (K). We restrict our result to be a cousin to avoid the pathological case where we follow many C edges, a Uk edge and many P edges and end up strictly above or below our starting place, i.e., where we pathologically create an ancestor, descendent or identity edge. In Figure 5A, cousin edge (f,g) is an edge in Uk. Our formula specifies that the cousin edges it induces are ((T,V), (S,V), (f,V), as well as the original edge (f,g). These are all the edges, induced by (f,g), which *Import* one part by another part; for example, induced edge (T,V) can be interpreted to mean that T imports (uses) V. (The reader may want to show that Uk in the above formula can be replaced by simply U and the formula will still yield the same result for Uk'.)

Induced Descendents and Ancestors. Here is the formula for induced descendents Ud':

Ud' = (Ao * Ud * Ao) ^ D +
(K * Uk * Ao) ^ D

The left operand of the union (+) induces edges from descending edges (Ud) while the right operand induces edges from cousin edges (Uk). In the left operand, we induce each Ud' edge by following any number of parent edges (Ao) then a descending explicit use edge (Ud), and then again any number of parent edges (Ao), but restricting the result to descendents (D). We restrict our result to be a descendent to avoid the pathological case in which we have created an edge which is an ancestor. In the right operand, we induce each Ud' edge by following a cousin edge K then an explicit cousin use edge Uk, and then any number of parent edges (Ao), but again restricting the result to descendents (D).

In graph G from Figure 5A there are no explicit descending edges (Ud is empty) as well as no ascending edges (Ua is empty too). As a result, so only Uk (explicit cousin usages) induces edges in this graph and the left operand contributes nothing. As an example, we will now explain how the right operand of the formula for Ud' specifies that cousin edge (f,g) induces descendent edge (V,g). Edge (f,g) induces (V,g) because V has (super) cousin (K) f, f explicitly uses cousin g (Uk), and g is its own (reflexive) ancestor (Ao); since (V,g) is a descendent edge (D), it is an edge in Ud'.

We now consider Ua', which has this formula:

Ua' = (Do * Ua * Do) ^ A +
(Do * Uk * K) ^ A

This is analogous to the formula for Ud', so we will not discuss it further.

Due to space limitations, we cannot give the algebraic specification for *hide,* which we use regularly in our Grok scripts, but instead we have given the formulae for Uk', Ud' and Ua' which arise from *hide* transformations. Since these are specified in an algebraic notation, we can manipulate them (for example, to prove that Uk can be replaced by U in the formula for Uk'), and we can mechanically interpret them.

One reason for presenting these formulae for induced edges in this paper is allow precise comparison of the conventions used by systems such as Rigi and PBS to display diagrams with hidden parts. Another is to help in the design of graph display systems which support hiding.

Consens [6] suggests diagrammatic notations that capture much of the power of our algebraic notation and offer the promise of a more easily readable, though less easily manipulable, notation.

5. Grok: A Relational Calculator

This section explains how formulae such as the ones given in the preceding section can be directly executed. An interpreter called Grok [9] has been developed which directly interprets binary relational operators such as relational composition and transitive closure; see also Relview [2] for another language that directly executes binary relational operators. The formulae given in the previous section can be directly executed by the Grok language.

Grok has input/output statements for reading *fact bases*, consisting of edges represented in

RSF notation. Here is a short but complete example Grok program:

```
getdb call.rsf
tcall := call+
putdb both.rsf
```

This program reads (gets) a data base in RSF, such as this one

```
call P Q
call Q R
```

from file *call.rsf*. Then it computes the transitive closure of the *call* relation, giving it the name *tcall*, and outputs both the *call* and *tcall* relations to the file *both.rsf*, as follows:

```
call P Q
call Q R
tcall P Q
tcall Q R
tcall P R
```

The output file contains the two edges of the *call* relation and the three edge of *tcall.*

The PBS system, which makes Grok available on the World Wide Web, provides Grok scripts that manipulate facts extracted from C programs to produce views like those in Figures 1 and 2. For example, we have used these scripts to manipulate parser-extracted facts from Linux source code to produce software architectural views of Linux. The parser we use for extracting facts from C programs produces facts at the low level of procedures and variables, but our visualizations are usually done at the file level and higher. We use Grok scripts to induce these low level facts up to higher levels.

For example, the PBS parser produces a relation called *sourcecall*, which records the fact that one function in a particular source compilation unit calls another function in that compilation unit. As well, there is a relation *funcdef*, which records the fact that a file has defined the body of a function, and a relation *funcdcl*, which records the fact that a file has declared the header of a function. We might have these three facts in RSF:

```
funcdef  main.c  main
sourcecall  main startup
funcdcl  start.h  startup
```

These mean that file *main.c* contains the body of function *main* which calls function *startup* whose header is in file *start.h.* The Grok script that reads these facts and induces the function-level call up to a file-level call uses a statement such as this:

```
scall := funcdef * sourcecall *
           inv funcdcl
```

This computes an *scall* edge for each path through a *funcdef* edge, then through a *sourcecall* edge, and then backwards (*inv*) through a *funcdcl* edge. This assignment statement to create *scall* is a special case of the general formula for Uk' given in the previous section. We use Grok to perform many other transformations, such as aggregating similar relations [11], which we do not have space to discuss in this paper.

Performance of Grok. Although Grok is a research prototype, it has been tuned to work acceptably well with fact bases of the size that are extracted from large industrial software. We have used Grok in reverse engineering a number of large systems (sizes all exceeding a quarter million lines of code). For example, the above formula to compute *scall*, which requires the evaluation of three relational operators (relational composition * twice and *inv* once), is executed by Grok in 1.0 seconds when operating on the fact base for the 300,000 line Cobol front end, running on a 5 year old Sun work station. A Grok script, such as the one to induce all procedure/variable-level facts up to the file level, runs in about a minute. These times are considerably less than the time to compile the source code for the system being analyzed. In the common case in which these scripts are run only as often as compiles, to produce standard views of the system, this speed is quite acceptable.

In the original naive implementation of Grok, we used $O(n^3)$ algorithms for operators such as relational composition, but in the current implementation we pre-sort operands, using a radix sort, and implement the operation as a merge, yielding an algorithm which is essentially O(e), where n is the number of nodes and e is the number of edges. If in the future we wish to use Grok to support an interactive system in which these transformations are done on the fly, as might be done in a system such as Rigi [12], the speed would need to increase by a factor of 10 to 100, which seems to be possible with new, faster hardware and further tuning of Grok's algorithms.

Using algebraic specifications and a language like Grok to execute them has several advantages. It allows us to state precisely what transformations we want in a notation that is declarative (when it is considered to be an algebraic formula). This allows us to separate our concerns about what transformations we require from how we are to implement them. This separation of concerns allows us to tune the general purpose implementation (the Grok interpreter) without the complication of considering any particular transformations. It implies that Grok is independent of the source language of the target software system. New source languages can be handled by writing the new Grok scripts, without changes to the Grok processor. A bonus is that Grok can be used independently from the rest of the PBS software; for example, it can be used to manipulate RSF before feeding it into the Rigi visualizer.

6. Conclusions

In this paper we have pointed out that certain structure transformations, such as hiding, are commonly carried out in the extraction of a software architecture during reverse engineering. We showed that the first, messy pictures, called Slonim diagrams, that we draw based on automatically extracted facts need to be transformed into more comprehensible diagrams. We introduced Tarski's binary relational algebra as a means of specifying these transformations. We showed that edge induction, in which higher level edges are created based on lower level edges, is a commonly occurring graph manipulation during reverse engineering. We showed how this induction can be precisely specified in relational algebra. We showed how specifications written in this algebraic way can be directly executed to help out us automate the transformation of huge numbers of facts into comprehensible views. (See [1] for an evaluation of PBS and Grok from a user perspective.)

We hope the we have shown the promise of using a high level notation, namely algebraic relational operators, in a practical, productive and efficient way to help us visualize large scale software. (See [3] for another algebraic approach to manipulation of reverse engineering information.) We hope that our approach to specifying structural transformations will help take some of the mystery out of the job of software architecture extraction and will help automate part of the extraction process.

Acknowledgements. Thanks to S. Mancoridis, V. Tzerpos and H. Fahmy for help in developing of these ideas and to the referees for their suggestions for improvements.

References

1. Armstrong, M., Trudeau, C., Evaluating Architectural Extractors, Working Conference on Reverse Engineering, Honolulu, Oct 1998.

2. Berghammer R., von Karger B., Ulke C., Relation-algebraic analysis of Petri nets with RELVIEW, in: Margaria T., Steffen B. (eds.): *Proc. 2nd Workshop on Tools and Applications for the Construction and Analysis of Systems* TACAS 96, Passau 1996, LNCS 1055, Springer: Berlin-Heidelberg-New York, 49-69 (1996).

3. Canfora, G., Cimitile, A., DeLucia, A., Specifying Code Analysis Tools, Nov 4-8, 1996, Monterey, Calif.

4. Chen, Yih-Farn, Nishimoto, Michael Y., Ramamoorthy, C.V., The C Information Abstraction system, IEEE Trans. on Software Engineering, vol 16, no 3, Mar. 1990, pg. 325-334.

5. Carmichael, I., Tzerpos, V. and Holt R.C., Design Maintenance : Unexpected Architectural Interactions, *Proc. International Conference on Software Maintenance*, ICSM '95.

6. Consens, M.P.,. Eigler, F. Ch., Hasan, M. Z., Mendelzon, A.O., Noik, E.G., Ryman, A. G. and Vista, D., Architecture and Applications of the Hy+ Visualization System *IBM Systems J*. 33:3 (1994), pp. 458-476.

7. Fahmy, H., Holt, R. C., and Mancoridis, S., Repairing Software Style Using Graph Grammars, *IBM Centre for Advanced Studies Conference*, Toronto, Canada, Nov, 1997.

8. Finnigan, P., Holt, R., Kalas, I., Kerr, S.,. Kontogiannis, K., Muller, H., J. Mylopoulos, Perelgut, S., Stanley, M. and Wong, K. The Software Bookshelf, *IBM Sys. Journal,* Vol. 36, No. 4, pp. 564-593, Nov 1997.

9. Introduction to PBS: The Portable Bookshelf, www.turing.toronto.edu/~holt/pbs/, 1997.

10. Bass, L., Clements, P., and Kazman, R, *Software Architecture in Practice,* Addison-Wesley, SEI Series 1997.

11. Kazman, R. and Carriere, S. J., View Extraction and View Fusion in Architectural Understanding, *Proceedings of ICSR* 5, Victoria, B.C., June 1998.

12. Muller, Hausi A., Mehmet, O.A., Tilley, S.R., and Uhl, J.S., A Reverse Engineering Approach to Subsystem Identification, *Software Maintenance and Practice*, Vol 5, 181-204, 1993.

13. Shaw, M., Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall 1996.

14. Tarski, A., On the calculus of relations, J. Symb. Log. 6, 3, 1941, pp 73-89.

15.Tzerpos, V. and Holt, R.C., A Hybrid Process for Recovering Software Architecture, *Proceedings of CASCON* (IBM Centre for Advanced Studies Conference), Toronto, Canada, Nov. 1996.