Software Architecture Abstraction and Aggregation as Algebraic Manipulations

Richard C. Holt

Dept. of Computer Science, University of Waterloo, Waterloo ON N2L 3G1 Canada
holt@uwaterloo.ca

Abstract. The structure of a large system can be made comprehensible if it can be viewed as a modest number of interacting entities. For example, the 800,000 line Linux kernel can be reasonably viewed as its five top-level interconnected subsystems. Each abstracted entity (subsystem) may have aggregated attributes, e.g., the total count of the number of lines of code in it, or the set of programmers who have worked on it. Given that the software architecture of the system is represented as a typed graph, both abstraction (ignoring details inside subsystems) and aggregation (collecting selected internal information) can be specified as algebraic manipulations. Our approach, based on Tarski relational algebra, allows direct execution of these abstraction and aggregation specifications by means of a relational calculator.

1 Introduction

Graphs have been used as the basis for reverse engineering large software systems such as Linux [2]. For example, Figure 1 gives a diagram of the top level of Linux, showing the system's five highest level subsystems. In the web-based display of these diagrams, supported by the PBS system [7], there is a diagram for the top level view as well as for each subsystem, each sub-subsystem, and so on, down to the leaves which represent code files. The user can navigate among these diagrams with mouse clicks.

This type of diagram, sometimes called a Software Landscape [6], consists of nodes and edges. The nodes represent software entities, such as subsystems and procedures. The edges represent interactions between these entities. There are a finite number of distinct *types* of edges, such as *Call* edges and data *Reference* edges. In other words the diagram represents a *typed* graph, where the types of edges include Call and Reference.

When re-engineering a big software system, one of the central challenges is to abstract away details that are confusing and to display only that information which is relevant to a particular view. This paper gives algebraic specifications of *abstractions* which hide the clutter in a diagram, and *aggregations*, which collect relevant information from the hidden parts.

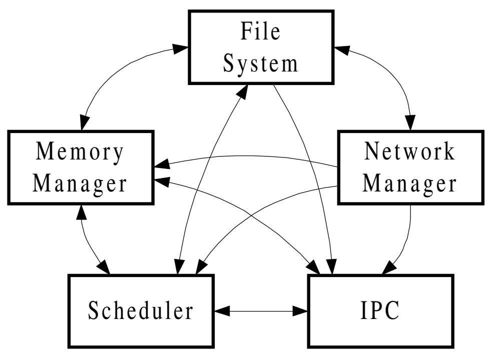

Fig. 1. Top-level view of Linux architecture. The boxes are subsystems and the arrows are procedure calls.

1.1 Creating Software Landscape Diagrams

The architecture of Linux was extracted [2] using the process illustrated in Figure 2. This process is largely automated by the PBS toolkit which is analogous to other software visualization systems such as Rigi [11], Dali [9], Krikhaar [5], and CIA [3] The PBS *extractor* (based on the Gnu C compiler) extracts facts, including the Call relationship, between procedures. The *hierarchy* is created largely manually and specifies the tree breakdown of the system into subsystems, recursively, down to atomic units, which may be code files, but could be lower level entities such as procedures and variables. The PBS *fact manipulator,* called Grok, emits a separate file of facts for each subsystem (and the whole system). The *layouter* determines (*x, y*) coordinates for each box (entity) so that the *visualizer* (a Java program) can display a separate diagram for each subsystem. The visualizer supports convenient direct manipulation of the diagram, for example, panning, zooming, and opening boxes.

The *facts* produced by the extractor are typed edges, e.g., the fact (*p* call *q*) means procedure *p* calls procedure *q*. The extractor also produces *attributes,* e.g., the *lineNo* attribute for procedure *p* gives the location of the declaration of *p* in the source code. In general, an attribute is a name with a value, e.g., the name *lineNo* could have as its value a file name and line offset, such as *main.c:27* (line 27 in the file *main.c*), or simply as a line offset, such as 27, if the file name is already known. In diagrams, relations are generally shown as arrows between boxes, while attributes are "popped up" upon user command. (We treat attributes as special kind of relation.)

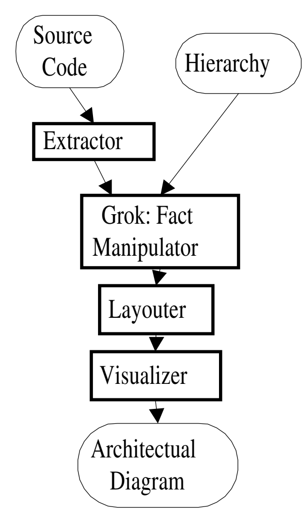

Fig. 2. Process of creating software landscape diagrams. The arrows show flow of information.

All these facts are represented externally in files in a simple ASCII exchange format known as TA (the Tuple-Attribute language). The TA language allows the specification of the *schema* for the facts; this schema is essentially the Entity/Relation Diagram for the facts. The TA language defines a general data exchange format for typed graphs; in other words, this format can be used outside of the PBS system for storing graphs in ASCII files.

1.2 Grok: A Relational Calculator

Grok is a relational calculator, using an in-memory data base, which inputs facts (relations and attributes) represented in TA, manipulates these, and emits the facts that are the result of this manipulation. Grok reads a script language (the Grok language) that specifies the manipulations to be carried out. As will be briefly explained, the Grok language is based on Tarski's binary relational algebra [12]. (For more details, see references [7] and [8].)

In the default operation of PBS, the parser emits 21 types of relations and a number of attributes. An initial PBS Grok script abstracts the facts up to the level of code files, thereby hiding procedures, variables and types. It also abstracts relations by combining similar kinds of relations, such as various kinds of calls to a single generic call relation. The successive Grok scripts further abstract the number of relations down to only three types, *useproc*, *usevar*, and *implementby*. (The implementby relation links a procedure header to its body.) The final set of Grok scripts separates out the edges specific to each subsystem and emits those edges to a separate files, in TA format.

Grok can be used independently from the PBS system. It is a general purpose "query engine", analogous to an RDB (relational data base), which supports graph queries, which are written as Grok scripts.

Since Grok is a general relational calculator, it is easy to modify the PBS Grok scripts to handle inputs from other parsers, including parsers for various programming languages, or to produce new kinds of views. This paper shows how this generalized approach can be used to create abstracted and aggregated views for viewing large software systems.

1.3 Overview of Paper

In the process of reverse engineering systems such as Linux, we have gained considerable practical experience in using and developing tools to create views of large systems. The successful use of this approach [8] had lead to user requests for aggregate attributes to accompany abstracted views. To a large extent, this paper is an attempt to clarify and formalize their demands.

In the next section, we will illustrate the kinds of *abstraction* we have used in Landscape diagrams, and the way attributes can be *aggregated* from hidden items up to the item that represents the hidden item. Then we will show how abstraction and aggregation can be formally specified using algebraic operations. We explain how Grok can execute these specifications. Finally, we give related work and our observations about this work.

2 Entity Abstraction and Attribute Aggregation

This section uses examples to introduce abstraction and aggregation. We will use the term *abstraction* to mean hiding details that are unimportant in a particular view. For example, in part (2) of Figure 3, the contents of node *v* are hidden (abstracted). We consider that *v* *represents* its hidden contents and that there is an *abstraction relation* α that links from v to its internal nodes, for example, α links *v* to *w* and *y* and *z*.

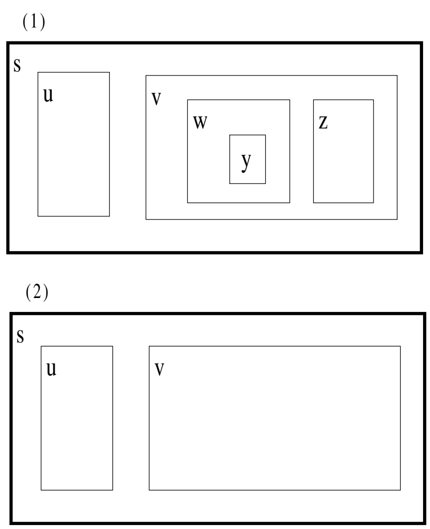

Fig. 3. Abstracting *v* and hiding its contents.

We are using the word "hiding" to describe abstraction, but be aware that what we are actually doing is more like "masking" or "condensing", in that the approach allows the hidden parts to be easily exposed as needed. When we are describing abstraction, we are describing a particular way to manipulate a graph. As we will show, these manipulations can be specified algebraically and carried out mechanically, thereby providing a great deal of flexibility in choosing exactly what views are to be produced.

Our examples will be extremely simple, to allow easy explanation, but please bear in mind that in actual use, we are commonly dealing with graphs that contain hundreds of thousands of edges.

In hierarchic (tree, nested) diagrams, such as the one in Figure 4, we will consider that there is a containment relation *C* from each containing box to its directly contained boxes. This is illustrated in Figure 4. As can be seen in the figure, the abstraction relation α locates the direct and indirect contents of *v*.

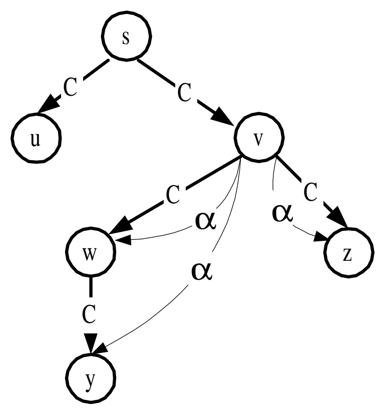

Fig. 4. The hierarchic containment relation *C* and the abstraction relation α for node *v.*

Figure 5 shows node *v* and its contained entities, along with a programmer attribute called *p*. For example, *Bob* is the programmer for node *v* while both *John* and *Betty* are programmers for *y*. The aggregate count of programmers of a node gives the number of programmers who worked on that node or any of its contents. For example v's attribute n_p is 4, because Bob, Alice, John and Betty worked on v and its contents. Note that Alice worked on two nodes (w and x). We say that n_p is an *aggregate* attribute because it is computed based on attributes of a node (v) and its contents. In particular, the aggregate attributes for node v depend on attributes in v together with attributes found by following abstraction edges α, from v to w, y and z. (In the figures, we show aggregate attributes, such as n_p, in parenthesis.)

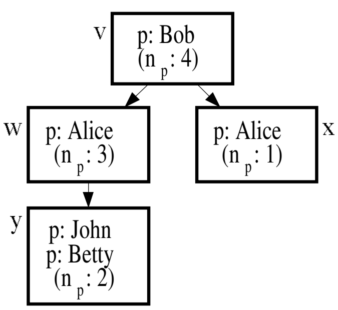

Fig. 5. Aggregate count n_p of the programmer attribute p.

Figure 6 shows another kind of aggregation. In the figure each node has a size attribute s; for example, v's size is 10 while y's size is 1000. The Σ_s attribute aggregates the total size of a node and its contents; for example, for v, Σ_s is 2510, which is the sum of 10, 500, 1000 and 1000 (the sizes of v and its contained nodes).

We have shown the abstraction of entity v, and we have given examples of two kinds of attribute aggregation: n_p (count of programmers) and Σ_s (sum of sizes). Next we will now show how *edges* can be abstracted and how their attributes can be aggregated.

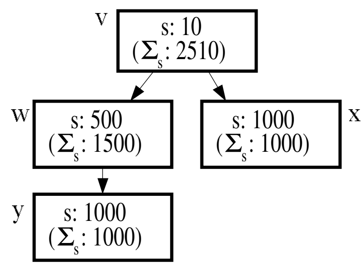

Fig. 6. Aggregate sum Σ_s of size .

3 Relation Abstraction and Attribute Aggregation

In viewing very large graphs, it is common to collapse several types of relations to a single type. For example, we may want to view both the Fetch and Store edges as Reference edges. Figure 7 gives a small graph in which procedure f fetches variable z (on line 10) and stores it (on line 15), while procedure g fetches it (on line 21). In this example, the Fetch and Store edges have a *lineNo* attribute, giving the source line number of the actual fetch or store.

Part (2) of figure 7 shows Fetch and Store abstracted into a single relation called Reference. It also shows that the lineNo attributes of Fetch and Store have been aggregated to corresponding Reference edges.

As well as relation abstraction by means of union (Reference is the union of Fetch and Store), it is useful to compose relations end to end. For example, in the C programming language, consider the case of a call from procedure body *f* to a procedure header (declaration) *h*. We will refer to this as an *hCall,* i.e., a call to a header. (See figure 8.) This header is implemented by a procedure body *g*. Part (2) of Figure 8 shows hCall and implBy composed (and abstracted) into Call. The Call relation links a procedure body (*f*) to a procedure body (*g*) that it is invoking. The two line numbers, 21 and 36, for the hCall edges have been aggregated to appear on the Call edge.

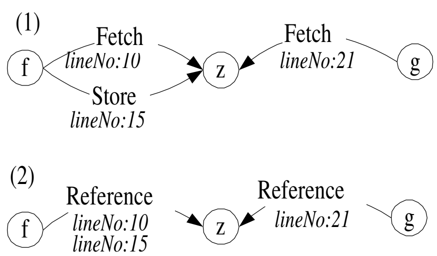

Fig. 7. Abstracting the union of *Fetch* and *Store* into *Reference.*

We have given examples of both entity and edge abstraction along with corresponding examples of attribute aggregation. Questions that naturally arise include: Is there a neat way to precisely define these kinds of abstraction and aggregation? Or, are they just ad hoc tricks illustrated by clever pictures? In order to answer this question, we need first to provide a mathematical basis, which we will now introduce.

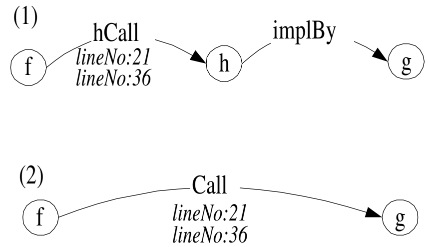

Fig. 8. Abstracting the composition of *hCall* and *implBy* into *Call.*

4 Tarski Binary Relational Algebra with Extensions

Tarski axiomatized a binary relational algebra in terms of operations including union $\cup$ and composition $\circ$ [12]. For example, suppose the Call relation contains edges (*p,q*) and (*p,r*), i.e., *p* calls *q* and *p* calls *r*, and the Reference relation contains (*q,v*), i.e., *q* references v. Then we can write

Depend = Call $\cup$ Reference

to mean that the Depend relation contains (*p,q*), (*p,r*) and (*q,v*). Similarly, we can take the intersection $\cap$ and the difference – of relations. In the following, if relation *R* contains pair (*x, y*), we write this fact as (*x R y*).

We define the inverse (or reverse) of a relation as all of its edges running backwards:

$R^{-1} = \{ (x,y) \mid (y\ R\ x) \}$

The relation composition operator $\circ$, as in

CallRef = Call $\circ$ Reference

creates the CallRef relation which follows one Call edge and then one Reference edge. In general, we define the composition of relations *R* and *T* as:

$$R \circ T = \{ (x,z) \mid \exists y : (x\ R\ y) \text{ and } (y\ T\ z)\}$$

We will consider that an attribute is a special kind of relation, namely, one which we choose not to draw as an arrow. This implies that, for example, relation R can be combined with attribute a by writing $R \circ a$.

The *id* (identity) operator, applied to a set of entities E, creates self loops on these entities:

$$id\ (E) = \{ (x,x) \mid x \text{ in } E\}$$

To the basic Tarski operators, we add transitive closure, which specifies one or more hops along the edges of a relation

$$R^{+} = R \cup R{\circ}R \cup R{\circ}R{\circ}R \cup \ldots$$

and reflexive transitive closure:

$$R^{*} = id \cup R^{+}$$

where *id* is

$$\{(x,x) \mid \text{for all } x \text{ in the graph}\}.$$

We will use # to mean cardinality, e.g., #{Bob, Alice}=2 and × to mean cross product, e.g., $\{A,B\} \times \{C,D\} = \{(A,C), (A,D), (B,C), (B,D)\}$.

We define the *projection* operator • in which an element s is pushed through a relation R:

$$s \bullet R = \{t \mid (s\ R\ t)\}$$

In addition, for set S, we define $S \bullet R = \{t \mid s \text{ in } S \text{ and } (s\ R\ t)\}$.

We have avoided the relational complement operator $\neg R$ and the complication that goes with it. However, much of the power of $\neg R$ can be simulated by subtracting relation R from the "universal relation" relation $E \times E$, where E is the set of entities in the graph.

We introduce a *for iterator* to take the union over a set of elements, e.g.,

$$\text{for } (\cup\ i \text{ in } \{2,3\} : \{4 * i\}) = \{4*2\} \cup \{4*3\} = \{8, 12\}$$

The definition of this iterator is:

$$\text{for } (\cup\ i \text{ in } \{i_1, i_2, \ldots\} : f(i)) = f(i_1) \cup f(i_2) \cup \ldots$$

We will use another version of this iterator that supports addition + instead of union ∪, e.g.,

$$\text{for } (+\ i \text{ in } \{2,3,4\} : 2*i\} = 4+6+8 = 18$$

The definition of this version of iterator is the same as the previous one, with ∪ replaced by +:

$$\text{for } (+\ i \text{ in } \{i_1, i_2, \ldots\} : f(i)) = f(i_1) + f(i_2) + \ldots$$

We will now show how Tarski's operators, along with extensions, can be used to specify abstraction and aggregation.

5 Entity Abstraction with Corresponding Attribute Aggregation

When we are abstracting, we pick a representative x which will represent a set of items $i, j, k, \ldots$ that are to be hidden. We link x to each of $i, j, k, \ldots$ by an abstraction edge α. In the case of entity abstraction, we will define α to be the transitive closure of the containment relation;

$$\alpha = C^{*}$$

It may seem surprising that we have used *reflexive* transitive closure here, especially since figure 4 does not show a self loop for node v. The reason we have done this is as follows. It allows us to aggregate across all of v's contents, as well as v itself. Using α defined this way, we can immediately specify v and its contents, using the projection operator •, as $v \bullet \alpha$. (In some kinds of abstraction, this reflexivity is not appropriate.)

Figure 9 specifies a number of ways to aggregate attribute a for the situation in which entity x is abstracted. Of course, there are other possible aggregations besides the ones we have listed there. The preconditions for these aggregations are given below along with details about them.

Union:
$U_{a,x} = x \bullet (\alpha \circ a)$
Distinct Count:
$n_{a,x} = \#\,(x \bullet (\alpha \circ a)) = \#\,U_{a,x}$
Inclusive Count:
$N_{a,x} = \#\,(x \bullet \alpha)$ (*Precondition 1*)
Sum:
$\Sigma_{a,x} = \text{for } (+\ i \text{ in } x \bullet \alpha : \text{elem } (i \bullet a))$ (*Precondition 1*)
Average:
$avg_{a,x} = \Sigma_{a,x} / N_{a,x}$ (*Precondition 2*)

Fig. 9. Aggregation of attribute a when abstracting entity x using abstraction relation α

The figure uses the notation elem(*S*) to extract the single element of singleton set *S*. Note that in the Sum aggregation, $i \bullet a$ in elem $(i \bullet a)$ is a singleton set.

The Inclusive Count and Sum aggregations require *Precondition 1*, which is as follows: each element located by α has exactly one attribute value *a*. The Distinct Count gives the number of all different abstracted attributes, while the Inclusive Count gives the number of all abstracted attributes, regardless of their uniqueness. The Average aggregation avoids zero division *by Precondition 2*, which requires that the count $N_{a,x}$ is non-zero. (When α is reflexive, Precondition 2 is necessarily true.) We will now illustrate the use of the aggregations given in Figure 9.

Figure 5 shows the programmers who worked on parts of a system. The abstractive relation α indicates that *v* is a representative for entities *w*, *y* and *z* (as well as itself). Union aggregation $U_{p,x}$ collects the names of programmers who have worked on *x* and its descendants.

For example,
$U_{p,v} = \{\text{Bob, Alice, John, Betty}\}$
$U_{p,w} = \{\text{Alice, John, Betty}\}$

We will show how to work out the value of $U_{p,v}$ to illustrate the use of the specifications given in Figure 9. We need the values of α and *p* in order to evaluate of $U_{p,v} = v \bullet (\alpha \circ p)$:

$$\alpha = C^* = \{(s,s), (u,u), (v,v), (w,w), (y,y), (z,z), (s,u), (s,v), (v,w), (w,y), (v,z), (s,w), (s,z), (v,y), (s,y)\}$$
$$p = \{(v,\text{Bob}), (w, \text{Alice}), (y, \text{John}), (y, \text{Betty}), (x, \text{Alice})\}$$

Using α we can evaluate $v \bullet \alpha$ as $\{v,w,y,z\}$. We can use this to evaluate the union of programmers for *v*:

$$\begin{aligned} U_{p,v} &= v \bullet (\alpha \circ p) \\ &= (v \bullet \alpha) \bullet p \\ &= \{v,w,y,z\} \bullet p \\ &= \{\text{Bob, Alice, John, Betty}\} \end{aligned}$$

We have used the identity here that: $v \bullet (\alpha \circ p) = (v \bullet \alpha) \bullet p$. Similarly, to compute the aggregate count of *v*'s programmers, we can evaluate:

$$\begin{aligned} n_{p,v} &= \#\,(v \bullet (\alpha \circ p)) \\ &= \#\{\text{Bob, Alice, John, Betty}\} \\ &= 4. \end{aligned}$$

To compute the aggregate sum of the size *s* of *v* (see Figure 6), we proceed as follows:

$$\begin{aligned} \Sigma_{s,v} &= \text{for } (+\ i \text{ in } v \bullet \alpha : \text{elem } (i \bullet s)) \\ &= \text{for } (+\ i \text{ in } \{v,w,y,z\} : \text{elem } (i \bullet s)\} \\ &= \text{elem } (v \bullet s) + \text{elem}(w \bullet s) \\ &\quad + \text{elem } (y \bullet s) + \text{elem}(z \bullet s) \\ &= 10+500+1000+1000 = 2510 \end{aligned}$$

We have illustrated how the aggregation formulae in Figure 9 can be used to determine the aggregate attributes of abstracting entities such as v.

These formulae show how to determine aggregate attributes of a particular, given entity *x*. They can be generalized to simultaneously determine the aggregates for all entities. For example, we can generalize the formula for the sum aggregate to define the union aggregate attribute Σ_a as follows:

Σ_a = for ($\cup$ x in ENTITY:
$\{x\} \times \{\Sigma_{a,x}\}$)

For every entity (ENTITY is the set of all entities of interest), this creates the pair (x, $\Sigma_{a,x}$); the set of all of these pairs gives the value of Σ_a. For example, for the graph consisting of only *v, w, y* and *z*, the result would be:

Σ_s = { (v,2510), (w, 500),
(y, 1000), (z, 1000) }

The generalized version of these formulae can be used to pre-compute the values of all aggregate attributes of interest.

6 Relation Abstraction

Figures 7 and 8 illustrate how edges can be combined and abstracted by relational union and composition. We will now specify the abstraction relation α for these two abstractions, and then we will show how the formulae in Figure 9 can be used to aggregate edge attributes. We will show how these formulae can be used to aggregate attributes, such as lineNo, of Store and Fetch edges, onto appropriate Reference edges. It is gratifying, and perhaps surprising, that we can use the same formulae (from Figure 9) to compute aggregation that arises from the abstraction of both nodes and edges.

In the case of relation abstraction, the abstraction relation α links edges to edges. For example, it links Reference edges to Fetch and Store edges. This means that we have extended the usual concept of graphs so that edges can act as nodes.

6.1 Aggregation Across Relation Union

We will first consider the case of relation *R* created as the union of relations *T* and *V*, that is, $R = T \cup V$. Figure 10 illustrates how abstraction relation α links *R* edges to *T* edges. In a similar way, α links *R* edges to *V* edges.

Figure 10 uses two new operators: the *tail* operator *t* and the *head* operator *h*. These operators allow us to in effect break an edge *e* = (*x R y*) into two sub-edges, namely its tail edge, which is ($x\ t_R\ e$) and its head edge, which is ($e\ h_R\ y$). These operations effectively create a new node *e*, whose label is "(*x R y*)". The tail relation t_R links from the starting entity *x* to *e*, while the head relation h_R links from *e* to the terminating entity *y*. We will use the *t* and *h* operators to specify the abstraction relation α.

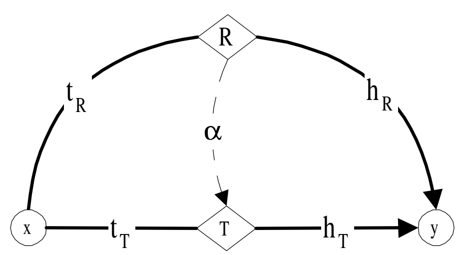

Fig. 10. Relation *R* represents (and abstracts) the union of *T* and *V*.

Each abstraction edge ($e\ \alpha\ f$) starts at *e*, which is an *R* edge, and ends at *f*, which is a *V* edge (see figure 10). We can specify those α edges that link *R* edges to *T* edges as follows:

$\alpha_T = (h_R \circ h_T^{-1}) \cap (t_R^{-1} \circ t_T)$

Looking at Figure 10, α_T creates edges that follow the head of an R edge (h_R), then follow backwards through the head of a *T* edge (h_T^{-1}); the result is actually an α edge only if it reaches the same *T* edge that can be found by following the *R* edge's tail backwards (t_R^{-1}) and then the tail of a *T* edge (t_T). This formula α_T gives all α edges that link *R* edges to *T* edges. In a similar way, the following formula gives all α edges from *R* edges to *V* edges:

$$\alpha_V = (h_R \circ h_V^{-1}) \cap (t_R^{-1} \circ t_V)$$

Combining α_T and α_R, we get all the α edges for the union of *T* and *V*:

$$\alpha = \alpha_T \cup \alpha_V$$

(Note that α is not reflexive in this definition.)

In the case of the graph in Figure 7, α_{Store} and α_{Fetch} are as follows:

α_{Store} =
{ ((f Reference z) α (f Store z)) }

α_{Fetch} =
{ ((f Reference z) α (f Fetch z)),
((g Reference z) α (g Fetch z)) }

In other words, Reference edge (*f, g*) represents Store edge (*f, z*) and Fetch edge (*f, z*), while Reference edge (*g, z*) represents Fetch edge (*g, z*).

Now that we have specified the abstraction relation α for relation union, we can use the formulae in Figure 9 to create aggregate attributes. We can use Figure 9's formula for union aggregation to create the union aggregation of line numbers corresponding to each Reference edge *x*:

$$U_{lineNo,x} = x \bullet (\alpha \circ lineNo)$$

For example, if *x* is the edge *e* = (*f* Reference *z*) in Figure 7, we can find its union aggregation of lineNo as follows:

$$U_{lineNo,e} = e \bullet (\alpha \circ lineNo) = \{10, 15\}$$

The interpretation of this is that Reference edge (*f, z*) corresponds to source code lines 10 and 15.

6.2 Aggregation Across Relation Composition

Figure 8 gives an example of combining edges by relational composition. We will now generalize this to the case of relations *T* and *V* composed (and abstracted) into relation *R*, that is: *R* = *T* o *V*. For this case, figure 11 shows that relation *R* is linked by the abstraction relation α to *T* and *V*.

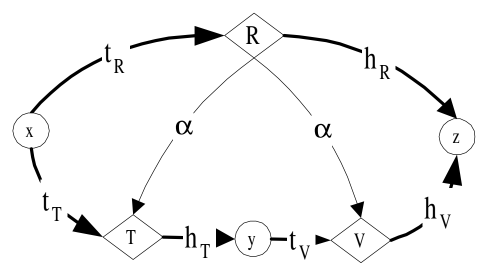

Fig. 11. Relation *R* represents (and abstracts) the composition of *T* and *V*.

Using an approach similar to that which we used for defining α for relation union, we separately define the contribution α_T of *T* to α and the contribution α_V of *V* to α:

$$\alpha_T = (t_R^{-1} \circ t_T) \cap (h_R \circ V^{-1} \circ h_T^{-1})$$
$$\alpha_V = (h_R \circ h_V^{-1}) \cap (t_R^{-1} \circ T \circ t_V)$$
$$\alpha = \alpha_T \cup \alpha_V$$

For example, in the graph in Figure 8:

α_{hCall} = { ((f Call g) , (f hCall h))}

α_{implBy} =
{ ((f Call g), (h implBy g))}

The interpretation of this is that the Call edge (*f, g*) represents the two edges (*f* hCall *h*) and (*h* implBy *g*).

Using the formula for union aggregation from Figure 9, we can find the lineNo attribute for edge *e* = (*f* Call *g*) as follows:

$$
\begin{aligned}
U_{lineNo,e} &= e \bullet (\alpha \circ lineNo) \\
&= (f\ Call\ g) \bullet (\alpha \circ lineNo) \\
&= ((f\ Call\ g) \bullet \alpha) \bullet lineNo \\
&= \{(f\ \ hCall\ h), (h\ implBy\ g)\} \\
&\qquad \bullet lineNo \\
&= \{21, 36\}
\end{aligned}
$$

The interpretation of this is that the call from procedure *f* to *g* corresponds to source code lines 21 and 36.

It is perhaps worth re-iterating the fact that in edge aggregation, we have used "second order" edges, i.e., edges from edges to edges. This second order concept arises naturally when one edge abstracts or represents other edges. We were able to model these second order edges by enhancing our graphs with nodes that correspond to edges, e.g., node *e*, with identifier "(*x R y*)" corresponds to edge (*x R y*).

7 Related Work

The Dali project [9], in which SQL queries are used for abstraction (which they call aggregation), is similar to our work on abstraction. The Rigi system [11 uses entity abstraction as well as "compound edges", which are similar to our concept of edge abstraction.

Feijs, Krikhaar et al [5] have taken a relational approach to software architecture, which is much like our own. They introduce a form of aggregation for relations with weighted edges. Their union operator for weighted edges provides edge count aggregation that is equivalent to that which we illustrate in Figure 7.

Although a considerable amount of research has gone into reverse engineering tools similar to PBS and Rigi, there is still only a modest use of these tools in industrial practice. It seems that we still need to perfect these tools, to make them easier to use, and to provide "value added" features, such as the aggregations described in this paper, for these tools to be widely accepted.

There are Tarski algebra based languages, besides Grok, such as Relview [1], but it appears that they have not been used to support software visualization and abstraction.

Clearly, SQL and standard data algebra can be used instead of Tarski's operators and Grok. The advantage of Tarski's operators is that they allow a more intuitive and elegant specification of manipulations such as aggregation, than is possible in notations such as SQL or Prolog. However, more research is needed to see which of these notations will prove to be most useful in practice.

Consens [4] introduces 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.

Abstraction and aggregation are clearly related to data mining. Miller's work [10] explores how data mining techniques can be used to help in large scale software understanding.

8 Conclusion

In this paper, we have shown how a formal approach based on Tarski relational algebra can be used to specify abstraction and aggregation. Using the Grok relational calculator, we have directly executed abstraction specifications to produce visualizations of large software systems such as Linux.

In particular, we have represented the facts about large software systems as large typed graphs. We have shown how Tarski's operators with extensions allow brief, formal, and executable specifications for abstraction and aggregation.

Our hope is that these specifications will help clarify some of the simplifying transformations, especially abstraction and aggregation, that are carried out in software re-engineering efforts. Since these specifications are directly executable, they hold out the promise of a declarative approach to the exploration of the huge amount of information that represents large software systems.

The contributions of this paper include the following: (1) pointing out that aggregation is an essential concept in reverse engineering of large software systems, (2) showing how certain aggregations can be concisely and formally specified using Tarski algebraic expressions, (3) specifying certain key aggregations that are useful in practice, (4) showing how these specifications can be directly executed, and (5) showing how the results can be fit into existing tool sets for visualizing large software.

Our formal approach to abstraction and aggregation may be of use in other areas of work such as data mining, beyond software re-engineering, in which large and complex structures need to be explored and understood.

Acknowledgements

I want to thank Mike Godfrey, Vassilios Tzerpos and Ivan Bowman for their careful reading of this paper and their valuable suggestions and corrections. The anonymous referees provided a number of valuable suggestions, which helped improve this paper.

About the Author

Ric Holt is a Professor of Computer Science at the University of Waterloo His internet address there is holt@uwaterloo.ca.

References

[1] 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).

[2] Linux as a Case Study: Its Extracted Software Architecture, Ivan T. Bowman, Richard C. Holt and Neil V. Brewster, accepted to ICSE 99, Los Angeles, May 99.

[3] 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.

[4] 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.

[6] Feijs, L.M.G., Krikhaar, R. L., and van Ommering, R.C., A Relational Approach to Software Architecture Analysis, Software Practice & Experience, 28(4) 371-400, April 1998.

[5] 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.

[7] Introduction to PBS: The Portable Bookshelf, www.turing. toronto.edu/pbs/, 1997.

[8] Structural Manipulations of Software Architecture using Tarski Relational Algebra, R. C. Holt, WCRE: Working Conference on Reverse Engineering, Honolulu, Oct 1998.

[9] Kazman, R. and Carriere, S. J., View Extraction and View Fusion in

Architectural Understanding, *Proceedings of ICSR* 5, Victoria, B.C., June 1998.

[10] R. J. Miller and A. Gujarathi. Mining for Program Structure. To appear in the International Journal on Software Engineering and Knowledge Engineering, 1999

[11] 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.

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