Binary Relational Algebra
Applied to Software Architecture

R. C. Holt, holt@csri.toronto.edu
Computer Systems Research Institute, University of Toronto

13 Mar 96, revised 5 June 96

CSRI Technical Report 345

Binary Relational Algebra Applied to Software Architecture

R. C. Holt, holt@csri.toronto.edu
Computer Systems Research Institute, University of Toronto
13 Mar 96, revised 5 June 96

Abstract. The *architecture* of a piece of software refers to its structure as given by its components and their interconnections. An *architectural style* is a set of commonalities among the structures of a number of software systems. We are concerned with architectures that are characterized by directed graphs. The graphs we are interested in contain edges that are "typed" or "colored", which means that there are a finite number of different kinds of edges. The nodes in the graphs represent entities such as procedures, modules and subsystems. The edges in the graphs represent relations among these entities, such as "calls", "accesses", "imports", "includes", and "uses".

This paper shows how binary relational algebra, as defined by Tarski and refined by Schmidt and others, can be used to formalize this type of architectural style. We show how this algebraic approach can be used to give constraints among the relations in an architecture. For example, we can use a relational inequality to formalize the rule that: items within an object can be used from outside the object only if they have been exported from the object.

We have developed a prototype "relational calculator", which reads data bases that represent sets of relations, and which allows us to compute new relations and sets. We have used this calculator in a reverse engineering project to determine the clients of a given subsystem, based on relations for containment and usage.

We formalize the concept of an architectural style in terms of rules that limit the use of relations. We propose a form for rules in which each rule is written as a relational inequality. We show that such rules are monotonic, which implies that adding new edges can allow more edges, but cannot disallow previously allowed edges.

Contents

1. Introduction

In recent years there has been a great deal of research involving software architecture. The work has arisen from the fact that the large scale structure of software must be well understood, or else the software is in danger of collapsing under the weight of its own complexity.

Garlan and Shaw [93] have characterized architectural styles as different ways of organizing the large scale structure of software. Perry and Wolf [92] have characterized underlying concepts in software architectures. Dean and Cordy [95] have presented a model of software architecture that is based on typed or colored graphs, together with graph grammars that determine the configurations of such graphs.

In this paper we are concerned with certain styles of software architectures. Like Dean and Cordy, we will be interested in software architectures that are modeled as directed graphs. Buxton [93] describes this approach, likening the style of a software architectural design to the constraints imposed by an entity-relation diagram [Chen 91].

This paper is organized as follows. We begin by defining colored graphs (those with more than one type of edge), and then use an example to illustrate how certain kinds of software architectures can be represented by these graphs. Then we present an example rule that constrains the use edges. We use this example rule to introduce ways of formalizing such rules. We introduction of relational algebra operators, notably relational composition, and show how they allow us to concisely state style rules. With these examples behind us, we introduce relational algebra. Next we show how the facts (tuples) in a colored graph can be represented as a simple data base, and how we have used a relational calculator, called Grok, to operate on such facts to aid in software reengineering. The next parts of the paper show how relational algebra is useful for stating permissions and requirements for software architecture and for formalizing certain architectural styles. Finally, a particular form of style rules, called the Sum-of-Products form, is useful. This form enforces a degree of simplicity in that the rules of this style are inherently "monolithic".

2. Colored (Typed) Graphs

We will be interested in finite graphs. We will often refer to the *vertices* of the graph as *nodes* or *boxes*. The nodes are interconnected by "colored" edges (or arrows or arcs).What we mean by "colored" is that there is a finite set of distinguished edge types (denoted by the colors) and that each edge has a particular type (color). In this paper we will not actually use colors, such as red and green, but in our practical software work, we commonly use colors.

Given a set of vertices V and relations R1, R2, ... Rn of colors 1, 2, ... n, each relation Ri consists of a set of ordered pairs of vertices:

$$Ri \subseteq V \times V$$

With this convention there can be several edges, of different colors, between particular nodes a and b.

3. An Example System Architecture

We will now give an example of a software architecture modeled as a colored graph. Figure 1 shows a software system that has four types (colors) of edges: I (import), E (export), U (use) and C (contain). As drawn on the left in Figure 1, the containment relation C is shown by means of nested boxes, as is common in architectural design diagrams. On the right, containment is shown directly by means of edges labeled C.

If we had colors available, we might color the C edges purple, the I edges black, the E edges green, and the U edges red. There are eight nodes: r (the root or main node), node a which contains nodes v, w and x, and node b which contains nodes y and z.

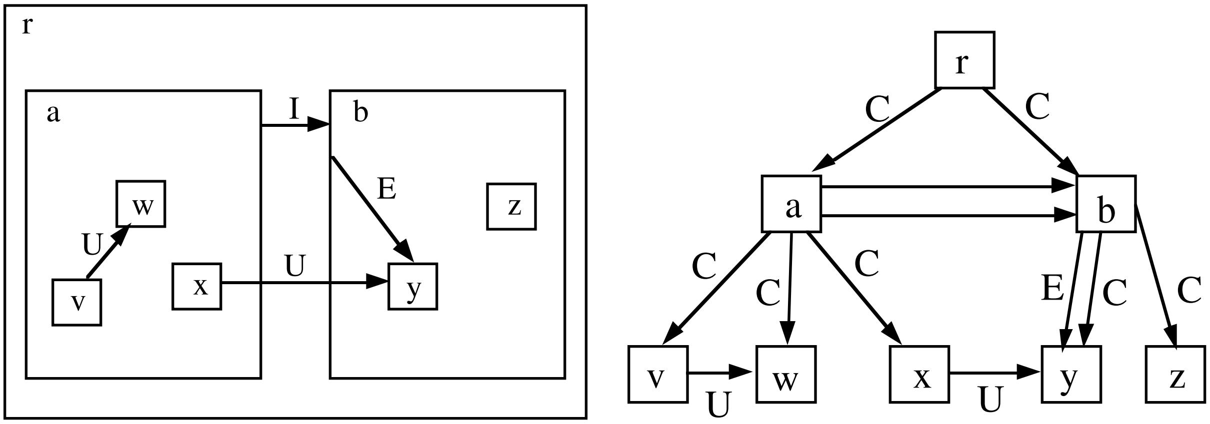

Figure 1. Example architecture, with eight nodes and four types of relations: I, E, U, and C.

More precisely, the set of nodes is:

V = {r, a, b, v, w, x, y, z}

and the relations are:

Import	I	= { (a,b) }
Export	E	= { (b,y) }
Use	U	= { (v,w), (x,y) }
Contain	C	= { (r,a), (r,b), (a,v), (a,w), (a,x), (b,y), (b,z) }

Aside. Some people prefer to draw an export edge to run, for example, from y to b instead of vice versa, to signify that y has become visible outside of b. We prefer to say that b exports y (or b gives permission for y to be seen), in which case it is natural to draw the arrow from b to y.

4. An Example Architectural Style Rule

In a system such as the one in Figure 1, it is common to use this rule:

Rule 1a. Module p can use module q if p and q are contained in the same subsystem or if module p's parent imports module q's parent and module q's parent exports q.

This is a simplified version of a rule used in languages such as Euclid [Lampson 77] and Turing [Holt 88], as well as in MILs (Module Interconnection Languages) [Prieto-Diaz 86].

The system in Figure 1 obeys Rule 1a. Module v is allowed to use (U) module w because both these nodes are contained in the same subsystem (a). Module x is allowed to use (U) module y because x's parent a imports y's parent b and b exports y. On the other hand, module w is not allowed to use module z, because z is not exported from b. Module z is not allowed to use any modules contained in module a, because z's parent b does not import a and because a does not export any of its children.

We say that the example in Figure 1 is *well-formed* according to Rule 1a, because it obeys this rule. Equivalently, we say that this example satisfies the *style* given by this rule. Later we will give formal definitions of the terms "style" and "well-formed".

We will now give definitions in order to formalize Rule 1a. We define the notation

$$x \; R \; y$$

to mean that (x,y) is in relation R. Sometimes we will refer to x R y as a *tuple*.

We define the S relation to mean "proper sibling", as follows:

Definition 1. S is the *proper sibling* relation. We define that x S y is true iff x and y are distinct nodes that have the same parent (iff there exists q such that q C x and q C y and $x \neq y$). We will shorten the term "proper sibling" to simply *sibling* in this paper.

S is a reflexive relation, i.e., x S y iff y S x. In our example, S contains (a,b), (b,a), (v,w), (w,v), (w,x), (x,w), and so on.

Definition 3 will introduce the parent relation P as the *inverse* of the child relation C. In general, the inverse (sometimes called the *converse*) of relation R, written as R^{-1}, consists of R's tuples reversed:

Definition 2. $R^{-1} =_{def} \{ (x,y) \mid y \; R \; x \}$

Definition 3. $P =_{def} C^{-1}$

In the right part of Figure 1, a parent edge x P y can be drawn whenever edge y C x appears.

We will now give a simplified version of Rule 1a using predicate calculus:

Rule 1b. $x \; U \; y \;\Rightarrow\; x \; S \; y \;\vee\; \exists\, m,n : \; x \; P \; m \;\wedge\; m \; I \; n \;\wedge\; n \; E \; y$

This rule is read as: x can use y only if x and y are siblings or if x has parent m which imports n which in turn exports y. (This is a simplified version of Rule 1a in that Rule 1b does not explicitly require n to be a parent of y. In a later section we will introduce a style, the Whole Import/Export style, that handles this circumstance.)

We can state Rule 1b more concisely by introducing the relational composition operator °. This operator effectively attaches, end to end, all edges of one color to edges of another color.

Definition 4. $R1 \circ R2 =_{def} \{ (x,y) \mid \exists\, z : \; x \; R1 \; z \;\wedge\; z \; R2 \; y \}$

For example, in Figure 1, the $I \circ E$ relation consists of the single edge (a,y), because I = {(a,b)} and E = {(b,y)}.

Since the composition operator ° is associative, we can write $(R1 \circ R2) \circ R3$ or $R1 \circ (R2 \circ R3)$ without parentheses as $R1 \circ R2 \circ R3$. Although it is common [Desharnais 94] to omit the ° operator, for example to write $R1 \circ R2 \circ R3$ as R1 R2 R3, in this paper we will explicitly write the ° operator to avoid possible confusion.

Rule 1b can now be written more succinctly as:

Rule 1c. $x \; U \; y \;\Rightarrow\; x \; S \; y \;\vee\; x \; P \circ I \circ E \; y$

Rule 1c uses the relation P°I°E:

$$P°I°E = \{ (v,y), (w,y), (x,y) \}$$

The P°I°E relation contains edges that follow a P edge, then an I edge, and then an E edge. For example, in Figure 1 we can start at node v, follow a P edge to a (P, which is the inverse of C, is not drawn in Figure 1), then follow an I edge to b, and finally follow an E edge to y; this implies that pair (v,y) is in P°I°E. More generally, P°I°E contains the three edges (v,y), (w,y), and (x,y). Rule 1c implies that any "use" edge U that is not between siblings is required to be one of the three edges in P°I°E.

We can now demonstrate the notational elegance of relational algebra; in particular, it allows us to write rules 1b and 1c in this concise form:

Rule 1d. $U \subseteq S \cup P°I°E$

This rule can be read as: relation U must be a subset of the union of the S and the P°I°E relations. It can also be read, probably more clearly, as: each pair (x,y) in U must be a pair in either S or P°I°E. This last way of reading the rule makes it obvious that Rules 1c and 1d are equivalent.

As Mancoridis [95] has shown in his Ph.D. thesis, rules such as 1d can be presented in a visual formalism, as illustrated in Figure 2. In this figure, the dotted lines represent edges that are allowed to be drawn if the solid lines are present. A U edge is allowed from x to y if there is an S edge from x to y, or if starting from x, you can follow a P edge, then an I edge, then an E edge to arrive at y.

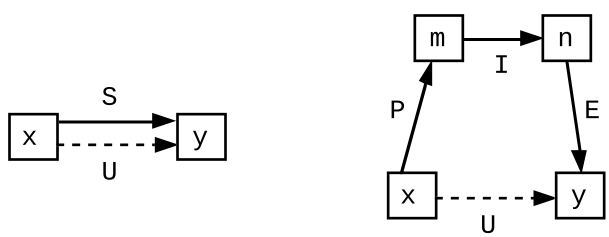

Figure 2. Rules 1b, 1c and 1d given in a visual formalism.

We have used an example (Figure 1) to illustrate how the architecture a software system can be modeled as a colored graph, and how relational algebra operators, such as relational composition, can be used to state rules (1a to 1d) of "well-formedness".

We have shown how colored graphes can be used to model the architecture of software systems, and how constraints on (rules about) the relations in these models correspond to restrictions we impose upon architectures. We will now turn to the question of formalizing our ideas about relations.

5. Binary Relational Algebra

The algebraic approach to binary relations, introduced by Tarski, provides axioms that define the effect of operators such as ° (relational composition) and relational union, along with the identity relation Id, the empty (or null) relation ϕ, and the full (or completely connected) relation L. (Our notation varies somewhat from other approaches. In particular, the letter I is commonly used to denote the identity relation. We have chosen to use Id for identity, thereby allowing us to use I as the name of common architectural relations such as "import" and "include".)

This mathematical approach then shows that these axioms have as a "model" a set theoretic interpretation in which a relation is a set of pairs from a given set. For our purposes, we will not be concerned with the distinctions between the "model" and its more abstract axiomatic theory. Interested readers should refer to works such as [Tarski 41, Schmidt 93] for details on a more formal approach to this subject. While many authors refer to this theory as simply "relational theory", we have called it "binary relational theory" because the theory of relational data bases, which uses n-ary relations, is so well known.

We will note here that in binary relational theory, equalities such as these are true:

$x \text{ Id } y$	$= (x = y)$	
$R \circ \text{Id}$	$= \text{Id} \circ R$	$= R$
$R \circ \phi$	$= \phi \circ R$	$= \phi$

The full relation L has the property that for any x and y, x L y is true. Besides relational composition, we have relational union, intersection, complement, inverse, subset, and difference:

$R1 \cup R2$, $R1 \cap R2$, $-R$, R^{-1}, $R1 \subseteq R3$, $R1 - R2$,

As well as the inverse of a relation, we define powers and closure of a relation:

R^n	$=$ if $n = 0$ then Id else $R \circ R^{n-1}$	n hops along R edges
R^+	$= R^1 \cup R^2 \cup R^3 \ldots$	One or more hops along R edges
R^*	$= \text{Id} \cup R^+$	Zero or more hops along R edges

We also define set projection, which effectively transmits the members of a set S through a relation to determine another set:

$S \circ R$	$= \{ y \mid \exists x \text{ in } S : x R y \}$	Forward projection of set S
$R \circ S$	$= \{ x \mid \exists y \text{ in } S : x R y \}$	Backward projection of set S

We could have used any of a number of formal approaches, other than binary relational algebra, which would be sufficient for formalizing our architectural models. For example, we could have used: Datalog (formalized SQL) [Ullman 88], Prolog, Hy+ visual queries [Consens 94], predicate calculus, or graph grammars [Blostein 96]. However, we have chosen to concentrate on binary relational algebra because many of its operations correspond closely to manipulations that are commonly used when dealing with software architectures.

6. RSF: Storing Relational Facts in a Data Base

We can think of relations in terms of a formal algegraic system, as we have just discussed, but we can also think of relations as a kind of data base with an associated calculating machine, as we will now explain.

From the Rigi Project [Muller 93] we have adopted a simple and highly effective notation, called RSF (Rigi Standard Form), for representing "facts" about a piece of software. Each of these facts is a pair in a relation. For example, we could have the pair (polyn,sqr) in relation "call", which we can take to mean that procedure polyn calls procedure sqr. In RSF, this fact would be written as:

```
call        polyn  sqr
```

In general, in RSF a tuple x R y, is written as three blank-delimited strings on a line, with the relation name R first, then x and then y. Here are some other facts written in RSF:

```
call        sort  swap
declare     main  x
```

```
declare     main  y
import      compiler  stack
include     main.c  io.h
export      stack  pop
own         "George Walker"  parser
```

A list of such facts can conveniently be stored in a computer file. Such a file constitutes a data base (or fact base). We use programs called *fact extractors* that read other programs and automatically produce corresponding data bases. Such data bases can be read by Rigi software, as well as by Landscape [Mancoridis 94] software, both of which automatically produce visualizations similar to Figure 1.

7. Grok: Implementing Binary Relational Calculus

We have implemented a system called Grok which is a calculator for relational algebra expressions. A person using Grok can give interactive commands to read and write RSF data bases and to do calculations using operators such as relational composition, closure and subtraction. The results of these operations can be assigned to new variables which are relations. The computed relations can be displayed on the screen and can be written to a file as a part of an enlarged data base.

Grok supports sets as well as relations. The members of the sets are strings, such as "sort", "main", "io.h", and "George Walker". Grok supports set operations, such as union, intersection and subtraction. Grok supports specialized operations, such as finding the subset of a given set whose names begin with a particular prefix. There is also a *slice* operation that removes from the data base all tuples that do not refer to members in a given set S. Grok also supports the set projection operations S ° R and R ° S.

We will now give an example session using Grok. In this example, the user gives a command to read a data base from a file called call.rsf. This data base contains tuples for the call graph of a large software system. Next the user computes the callers and callees of a procedure named ds.pl9. This is essentially a real example from an on-going reengineering project, in which ds.pl9 is the Dead Store elimination procedure (ds) written in the language PLIX (pl9). Grok uses the ASCII character * instead of ° for composition and projection. Grok treats a string such as ds.pl9 as a singleton set.

`getdb call.rsf`	Read RSF data base from the file call.rsf
`ds.pl9 * call`	Calculate set of procedures that ds.pl9 calls
...Grok prints list of procedures that ds.pl9 calls...	
`call * ds.pl9`	Calculate set of procedures that call ds.pl9
...Grok prints list of procedures that call ds.pl9 ...	
`cds = call * ds.pl9`	Calculate set variable cds of procedures that call ds.pl9
`cds * call`	Calculate set of procedures that cds calls
...Grok prints list of procedures called by procedures in cds ...	
`transcall = call+`	Compute the transitive closure of the call relation
`ds.pl9 * transcall`	Calculate set of procedures that ds.pl9 calls transitively
...Grok prints list of directly or indirectly called procedures ...	

In this session, Grok has been used interactively to explore the data base, in order to find out information related to a particular entity called ds.pl9.

8. Using Grok as a Programming Language

Besides being an interactive tool, Grok is a programming language in that scripts of Grok commands can be submitted to the Grok processor to carry out useful computations. For example, in our reverse engineering project, we often need to determine the clients, suppliers, etc., for a given subsystem ss:

Clients:	What entities (procedures) outside of ss use entities (procedures and variables) in ss?
Exports:	What entities in ss are used outside of ss?
Suppliers:	What entities outside of ss are used by entities in ss?
Buyers:	What entities in ss use items from outside of ss?

These four sets are illustrated in Figure 4.

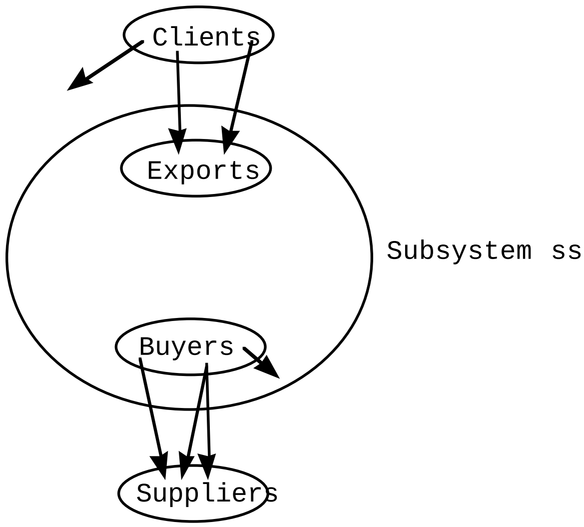

Figure 3. Clients, exports, suppliers and buyers for subsystem ss.

The relational calculations to compute these four sets, given set ss and given the use relation U, are:

CL = U ° ss - ss	**Clients**: Entities that use ss, not including entities in ss
EX = CL ° U ∩ ss	**Exports**: Entities that SS's clients use that are in ss
SP = ss ° U - ss	**Suppliers**: Entities that ss uses, not including entities in ss
BY = U ° SP ∩ ss	**Buyers**: Entities that use SS's suppliers that are in ss

Note that composition ° has higher precedence than intersection ∩ or difference -.

The current implementation of Grok is a prototype. It has been extensively used in computing information about a large IBM program that we have been reverse engineering, but it is not currently stable enough to be used by people other than its implementers. It imposes inconvenient restrictions, such as allowing only one operator in a given expression, and omits operators that have not yet been found to be needed for our reverse engineering work. From the experimental work we have done, we conclude that Grok-like processors have considerable promise in the area of software reengineering. It may well be that such processors could have a wide range of useful applications.

We now turn back to the subject of using relational expressions to impose constraints on software architectures.

9. Relational Permission, Requirement and Induction

In Section 3 of this paper, we showed how a relational inequality (Rule 1d) can be used to specify constraints on a system. In this section we will show how relational inequalities (and equalities) provide a succinct notation for writing down recurring ideas in software architecture.

In particular, we will discuss the ideas of *permission* for, *induction* of and *requirement* for relations. This will be done in term of these three (in)equalities:

Formula 1.	$R0 \subseteq P \circ R1 \circ C$		Permission for R0 to exist
Formula 2.	$R1 \supseteq C \circ R0 \circ P$		Requirement for R1 to exist
Formula 3.	$R1 = C \circ R0 \circ P$		Induction of R1 from R0

As before, we will assume that relation C (containment) is a tree and that relation P (parent) is the inverse of C. These three formulae (1, 2 and 3) can all be illustrated by a single diagram; see Figure 4.

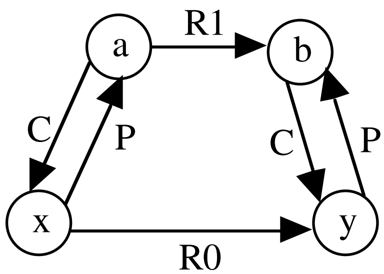

Figure 4. Illustration of Formulae 1, 2 and 3.

(1) Permission. Formula 1 can be interpreted to mean: edge x R0 y is *permitted* to exist if there exist edges x P a, a R1 b and b C y. In other words, x is permitted to interact with y if x's parent interacts with y's parent.

Formula 1 is similar to formulae such as

$$I \subseteq P \circ I \circ E$$

which is part of Rule 1d. In the case of Rule 1d, relation I (import) plays the role of both high level permission (R1) and low level usage (R0). Relation E (export) plays the role of C, because export gives permission to access a child. More generally, Formula 1 and variations of it can be used to specify import-like and export-like permissions. For example, Schwanke [89] describes a system in which the parts of two subsystems can be related by a particular kind of edge only if the parents have an edge of the same relation between them. His rule can be precisely specified as this version of Formula 1: $R \subseteq P \circ R \circ C$.

(2) Requirement. Formula 2 can be interpreted to mean: edge a R1 b is *required* to exist if there exist edges a C x, x R0 y and y P b. In other words, a and b are required to interact if their children interact.

Formula 2 can be thought of as the converse of Formula 1. Whereas Formula 1 records the permission that parents give to their children, Formula 2 records the requirements (or demands) that children place upon their parents. To extend this analogy: parents *permit* their children to eat food, and conversely children *require* (or demand) food from their parents.

(3) Induction. Formula 3 is used to *induce* (or define) higher level relations based on lower level relations. For example, if procedure p1 contained in file f1 calls procedure p2 contained in file f2, we may induce the "fact" that f1 calls f2. (This induction can continue through layers; for example if f1 is contained in subsystem ss1 and f2 is contained in ss2, we may induce that ss1 calls ss2.) We can write down this idea precisely as:

$$\text{CALL1} =_{\text{def}} \text{C} \circ \text{CALL0} \circ \text{P}$$

which is a version of Formula 3. This can be interpreted to mean: f1 is considered to call f2 if f1's child p1 calls f2's child p2.

Murphy and Notkin [95], in their article on Reflexion Systems (used in reengineering), induce a high level of "call" among subsystems based on procedure calls that span these subsystems. They formulate this concept in terms of the Z notation. While their formulation is correct, Formula 3 is more concise and, we suggest, easier to follow, at least for those who are familiar with relational algebra. Besides, Formula 3 is directly executable by a processor such as Grok, thereby allowing immediate inspection of the induced edges.

Formula 3 can be thought of as recording the balanced situation in which parents give their children all the permission their children need (and no more) and the children need all the permission their parents give (and no more).

Aside. Formula 3 defines "bottom up" induction in that higher level relations are induced from lower level relations. There is a form of "top down" induction that uses the formula $R0 = P \circ R1 \circ C$; the induced relation R0 is the maximal set of edges that lower level relation R0 is permitted, based on a higher level relation R1.

The formulae given in this section can be used in a system such as Grok to compute induced sets, and as well to see if the associated rules are being satisfied.

10. Systems and Styles

We will now give a framework for formalizing a software architecture and its style using relational algebra. We will not attempt to deal with the rich set of architectural styles presented by Garlan and Shaw, but will rather concentrate on what might be called MIL styles, in which the main concern is to decompose the system into parts and to control the interactions among the parts.

Definition 5. A *system* (or architecture) A is characterized by a non-empty, finite set of vertices V, a containment tree C, and a set of relations:

$$A = (V, C, R1, R2, \ldots Rn)$$

where C is a tree containing all vertices in V and

$$R1, R2, \ldots Rn \subseteq V \times V$$

Definition 6. We define the *parent* P relation and *sibling* S relation based on C:

$$P =_{\text{def}} C^{-1}$$
$$S =_{\text{def}} P \circ C - Id$$

Definition 7. We characterize the *style* of a system by giving a rule set, called RULE, which the system is to satisfy:

RULE = (RULE1, RULE2, ... , RULEx)

Each rule places constraints on the relations R1, R2, ... Rn. Rule 1d from Section 3 is an example of a rule.

Definition 8. System A is *well-formed* and we write WF(A, RULE) iff A satisfies all the rules of rule set RULE. When the rule set is obvious, we write simply WF(A).

We sometimes say the system is *legal* if it is well-formed. A system that is not well-formed is said to be *ill-formed* or *illegal.*

11. Examples of Styles

We will now give a number of rule sets that characterize various architectural styles:

Style 1. Whole Import/Export Style (for import I, export E and use U):
$(E \subseteq C,\ \ I \subseteq S \cup P°I°E,\ \ U \subseteq S°E^* \cup P°I°E^*)$

Style 2. Selective Import/Export Style (for import I, export E and use U):
$(E \subseteq C°E^*,\ \ I \subseteq S°E^* \cup P°I°E^*,\ \ U \subseteq S°E^* \cup P°I°E^*)$

Style 3. Tube Style (for tube T):
$(T \subseteq S \cup P°T \cup T°C \cup P°T°C)$

Style 4. Buyer/Seller Export Style (for export to buy B, export to sell E and use U):
$(B \subseteq C°B^*,\ \ E \subseteq C°E^*,\ \ U \subseteq B^{-1*}°S°E^*)$

In these rule sets, we assume that the names of the relations, R1, R2, ... Rn, in the architecture correspond to the names of relations in the rules, such as I, E and U.

Style 1, which has three rules, is a generalization of the style suggested by Rule 1d. In Style 1, a node can export (E) its children, and it can import (I) its siblings as well as the entities exported from its parent's imports. It can use (U) its siblings and their exports (through any number of levels). It can also use (U) the exported items (any number of levels of export) of its parent's imports. Style 1 is similar to the scope rules used by the Euclid and Turing languages.

Style 2 is a generalization of Style 1 that makes it more flexible and hence more suitable for MILs. It permits the importing or exporting of only a part of an entity. It also has the attractive simplicity that the right-hand side of the import (I) rule, $S°E^* \cup P°I°E^*$, is identical to the right-hand side of the usage (U) rule. As a result, usage (U) can be considered to be a special case of importing (I).

Style 3 attempts to be both simple and general, in that it consists of a single relation, the tube (T), which represents many of the combined concepts of importing, exporting and usage. In this style, a node can have a tube to its siblings, to entities its parents have a tube to, to the children of entities it has a tube to, and to the children of entities its parent has a tube to.

Style 4 supports two kinds of exporting. The first, B (export to buy), allows a node to use (to buy) entities that are the siblings of its parent (it can, in general, use the siblings of any of its ancestors, if they provide B access permission). Not only that, but of these usable entities, it can use any items that these items export to sell (E), to any level. This style is interesting in that it has no concept of importing, but instead two symmetric concepts of exporting.

Mancoridis [95] has shown how style rules such as those as Styles 1 to 4 can be formalized in a visual notation that is much like Figure 2.

12. Sum-of-Products Rules

Each of the rules that define Styles 1-4 has the form

$Ri \subseteq P1 \cup P2 \cup \ldots \cup Pm$ *Sum* where each Pj has the form

$Pj = Pj1 \circ Pj2 \circ \ldots \circ Pjq$	*Product*

We call this the *Sum-of-Products* or *SoP* form for rules, where we consider the unions $\cup$ to be sums and the relational compositions $\circ$ to be products. In the SoP form, we require each Pjk to be one of C, P, S or Ri, possibly operated on by closure (+ or *) or inverse. As an example, the second rule in the Selective Import/Export style:

$$I \subseteq S \circ E^* \cup P \circ I \circ E^*$$

is the sum of the two terms $S \circ E^*$ and $P \circ I \circ E^*$, where the first term is the product of S and E* and the second term is the product of P and I and E*.

The SoP form is simple and yet is sufficiently powerful to express many or perhaps most of the style rules for MIL-like systems. Mancoridis' visual notation is more powerful, in that it incorporates negation. He uses this power to effect in describing rather complex styles, such as the scope rules of the C++ language. While his notation has the advantage of allowing convenient specification of certain constructs, ours has the advantage of easier formal analysis. It remains to be seen if the extra power gained by negation is important when dealing with problems in software architecture.

One important property of SoP rules is that they allow us to give an alternate definition of well-formedness, which is based on individual edges. We start by defining when an edge is well-formed.

Definition 9. For a given set of SoP rules RULE, edge x Ri y is *well-formed* in architecture A iff there exists j such that x Pj y, in other words:

$$WF(x\ Pi\ y, RULE) =_{def} x\ Pi\ y \vee x\ P2\ y \vee \ldots \vee x\ Pm\ y$$

When the rule set is obvious, we write simply WF (x Pi y).

This definition says that a particular edge is well-formed if it is allowed by one of the terms of one of the style rules. If an edge is not well-formed, we say it is *ill-formed* or *illegal*.

Lemma 1. For a given set of SoP rules, system A is well-formed iff for all edges x Ri y in A, the edge is well-formed. In other words,

$$WF(A) \quad \text{iff} \quad \forall x,y,i : x\ Ri\ y \Rightarrow WF(x\ Ri\ y)$$

This lemma follows from the observation that the requirements to satisfy each SoP rule can be rewritten in predicate calculus as

$$\forall x, y : x\ Ri\ y \Rightarrow x\ Pi\ y \vee x\ P2\ y \vee \ldots \vee x\ Pm\ y$$

Lemma 1 allows us to consider edges one at a time when determining if a system is well-formed.

13. Monotonicity of SoP Rules

One of the key simplifying properties of SoP rules is the following: Adding an edge to a system can never cause other edges to become ill-formed. And new edge e can permit other new edges to be added to the system, but it can never prevent previously permitted edges from being added. We will formalize this property in terms of "monotonicity". We let f be a function of R1, R2, ..., and Rn. For convenience, we will let R represent all of these relations:

$$R =_{def} (R1, R2, \ldots, Rn)$$

so f is a function of R.

Function f could also be considered to be a function of C (as well as P and S), but we will consider that C is held constant, so we will not consider that C is one of f's arguments.

The right-hand side of each SoP rule is a function f. For example, the function for the right-hand side of the second rule of the Selective Import/Export style is

$$f(I, E, U) = S^\circ E^* \cup P^\circ I^\circ E^*$$

The idea that increasing the number of edges in a system can only provide more permissions (never less), is captured by the concept of monotonicity.

Definition 10. Function f is *monotonic* iff

$$(\forall\ i : 1 .. n\ :\ Ri \subseteq Ri')\ \Rightarrow\ f(R1, R2, ... Rn) \subseteq f(R1', R2', ... Rn')$$

Definition 10 implies that as you add edges to Ri, to change Ri to Ri', the set of edges determined by function f is non-decreasing.

Lemma 2. The right-hand side f of any SoP rule is a monotonic function.

Lemma 2 follows from the observation that adding an edge x Ri y to any relation Ri can only increase the number of edges allowed by each term Pj on the right-hand side of each SoP rule.

18. Conclusions

Our purpose in this paper has been to propose the use of binary relational algebra as a mathematical basis for studying certain kinds of software architectural style. While we could have used another foundation, such as predicate calculus, it appears that the succinctness and expressiveness of binary relational algebra is particularly well suited for this kind of work.

While we have used a number of diagrams in this paper, we have not concentrated on visual formalisms such as Mendelzon's Hy+ [Consens 94] data query language or Mancoridis' [95] notation that deals with binary relations. These visual formalisms provide the advantage that they bring the mathematical formalism close to the mental images that many people visualize.

We have used relational algebra to formalize several architectural styles. These styles seem particularly well suited for representing MIL dependency constraints.

We defined a class of styles, called Sum-of-Products (SoP) styles, which specify dependency constraints in software architectures. SoP styles are "monotonic", meaning that adding edges can increase, but never decrease, the permission for adding more edges.

It has been our goal in exploring this relational approach to software architecture to shed light on deeper questions such as: Is there a "right" way to formulate interdependencies in software architectures? Are there canonical styles that all students of software architecture should learn? Can architectures and architectural styles become mathematical objects subject to rigorous and useful analysis? While we have not definitely answered any of these questions, we hope that we have shed some light upon them.

Acknowledgments. This paper draws heavily on extensive interactions with Spiros Mancoridis during that last several years. It has also benefited considerably from suggestions by Vassilios Tzerpos, Eric Hehner and David Barnard. Ali Mili, Jules Desharnais, and Marc Frappier have introduced me to material on relational semantics.

References

Blostein, D., Fahmy, H. and Grbavec, A. , "Issues in the Practical Use of Graph Rewriting," Lecture Notes in Computer Science, Springer Verlag, to appear.
Buxton, John and McDermid, John, Architectural Design (Chapter 17), pp. 17/1 - 17/33, in Software Engineer's Reference Book, CRC Press, 2000 Corpoarate Blvd NW, Boca Raton, Florida, 33431, 1993.
Chen, Peter, The Entity-Relationship Approach to Database Design, QED Information Sciences Incl., 170 Linden St., Wellesley, Mass., 1977, 1991.
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), D. pp. 458-476.
Desharnais, J., Fairouz, T. and Khédri, R, Demonic relational semantics of sequential programs, Département d'informatique, Université Laval, Québec, QC, G1K 7P4 Canada, Dec 94
Dean, T.R., and Cordy, J.R. A Syntactic Theory of Software Architecture, IEEE Transactions on Software Engineering, Vol 21, No 4, April 1995.
Garlan, David and Shaw, Mary An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume 1, World Scientific Publishing Co., 1993
Holt, R.C. and Cordy, J.R, The Turing Programming Language, Communications of the ACM 31, 12 (December 1988), 1410-1423.
Lampson, B.W., Horning, J.J., Mitchell, J.G. and Popek, G.J. Report on the Programming Language Euclid, ACM SIGPLAN Notices 12, 2 (February 1977), 1-79.
Mancoridis, S., Holt, R.C. and Godfrey, M.W. Tool support for software engineering education, Proceedings of the ICSE-16 Workshop on Software Engineering Education (May 1994), pp 213-219.
Mancoridis, S., Controlling the Interactions of Architectural Design Components using Scoping Rules, PhD Thesis, Department of Computer Science, University of Toronto, Dec 1995.
Muller, Hausi A., Mehmet, O.A., Tilley, S.R., and Uhl, J.S., A Reverse Engineering Approach to Subsystem Subsystem Identification, Software Maintenance and Practice, Vol 5, 181-204, 1993.
Murphy, G.C, Notkin, D., and Sullivan, K., Software Reflexion Models: Bridging the Gap between Source and High-Level Models, Proceedings of the Third ACM Symposium on the Foundations of Software Engineering (FSE '95)
Perry, Dwayne E. and Wolf, Alexander L., Foundations for the Study of Software Architecture, ACM SigSoft, Software Eng. Notes 17,4, Oct 92, pp 40-52.
Prieto-Diaz, R. and Neighbors, J. M., Module Interconnection Languages, Journal of Systems and Software, Vol. 6, pp 307-334, 1986.
Schmidt, G. and Strohlein, T., Relations and graphs, EATCS Monographs in Computer Science, Springer-Verlag, Berlin, 1993.
Schwanke, R. W., Altucher, R.Z., Platoff, M.A., Discovering, Visualizing, and Controlling Software Structure, Proceedings of Fifth International Workshop on Software Specification and Design (Pittsburgh, Penn. May 1989), pp. 147-150.
Tarski, A., On the calculus of relations, J. Symb. Log. 6, 3, 1941, 73-89.
Ullman, J.D., Principles of Database and Knowledge-Base Systems (Volume 1), Computer Science Press, New York, New York, 1988.