Permission Theory for Software Architecture Plus Phantom Architectures

Richard C. Holt
School of Computer Science, University of Waterloo,
200 University Avenue West, Waterloo, Ontario N2L 3G1
1 519 888 4567
holt@uwaterloo.ca

ABSTRACT
Using reverse engineering techniques, a model of the architecture of a large software system can be recovered from its source code as a large hierarchic graph. In our approach, the hierarchy in the model is represented as a tree, whose nodes correspond to the parts and subparts of the system, such as subsystems, packages and classes. The *dependencies* between these parts, such as calls to procedures, are represented as directed edges from node to node. *Permission* edges can be added to the graph to document which dependencies are legal in the architecture. For example, a *public* edge from a module to one of its procedures could specify that calls are permitted to the procedure from outside the module. It is shown that certain of these architectural models are *phantoms, i.e.,* they legal, satisfying the permission rules, but cannot be built up out of smaller graphs that satisfy the rules.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – graph model, scope control, dependencies.

General Terms
Design, documentation, theory.

Keywords
Software architecture, software model, box and arrow diagrams, phantom, graph model.

1. INTRODUCTION
Over the last couple of decades, software reverse engineering techniques have been developed that take large software systems and create corresponding architectural models for the system [7, 12, 8]. These models can be used for various software engineering purposes such as program comprehension, maintenance, locating anomalous constructs, migration, etc.

For example, Bowman [1] carried out reverse engineering of the Linux operating system and described the upper hierarchic structure of its main subsystems, illustrated in Figure 1.

The root of the system, called Linux in Figure 1, contains 7 subsystems: Initialization, Memory Manager, Process Scheduler, etc. In turn, subsystems contain subsystems, for example, the File System contains Executable Formats, Virtual File System, File Quota and Buffer Cache, while Virtual File System contains Device Drivers and Logical File Systems. We have simplified Figure 1 by showing only part of the hierarchy. Not shown in Figure 1 are edges which represent static dependencies among the parts; these are shown in other kinds of visualizations of the graph model [9].

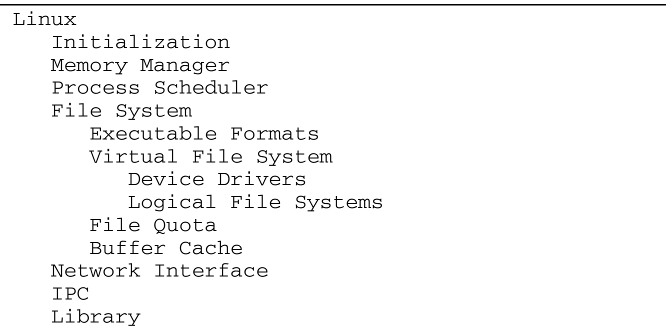

Figure 1. High level hierarchic structure of Linux.

This paper concentrates on a graph model of software architecture in which nodes represent constructs such as classes, modules, packages and subsystems. However, the actual model is mathematical in nature, so it could be used to represent hierarchic structures in other domains. A designated set edges of in the graph represent the system's nesting (hierarchic) structure while others represent dependencies (such as procedure calls and references to variables). The graph model can contain *permission* edges, representing constructs such as "import" of packages.

In this paper, we present a way in which permission edges in the graph can control access (or scope) among the parts of the system. This control is analogous to directives such as "private" and "protected" that limit access to parts of classes in Java and C++. This control can be used in higher levels in the system, for example, in Linux to prevent the Library subsystem from using other subsystems (see Figure 1) or to prevent all subsystems (except Initialization) from calling functions within the Initialization subsystem.

Based on a graph model, we define a notation that a system architect can use to write a *ruleset* that determines the meaning of permission edges in the target software system. For example, the architect might specify that a "public" edge from a subsystem S to a file F contained in S allows F to be accessed from outside S.

This paper gives examples of such rulesets and a general form for them, called *Sum of Products* (*SoP*). It is shown that SoP rulesets have two interpretations, which are often equivalent: (1)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'04, Month 1–2, 2004, City, State, Country.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

interpretation as formulas that directly determine if a graph is "legal" and (2) "constructive" interpretation that specifies how edges can be legally added to the graph. The surprising observation is that some graphs, called *phantoms,* are legal by the first interpretation, but are not *constructive* (cannot be built using the second interpretation).

The rest of the paper is organized as follows. The next section describes the basis of our model: Nested Box and Arrow (NBA) graphs. Relational operators that can query and manipulate these graphs are reviewed, and these are used to define "family" relations in NBA graphs. Then examples of permission rules are given. The general form of these rules is formalized as Sum of Products rulesets. The approach is generalized as Abstract Permission Theory, which is independent of the structure of NBA graphs The concept of phantom architectures is defined and explored. Practicality and related work are discussed, and finally conclusions are given.

2. NESTED BOX AND ARROW GRAPHS

We will use directed graphs, namely NBA graphs [5], with labeled (also called colored or typed) edges, to model the hierarchic architecture of software systems (see Figure 2). The hierarchy is defined by a set of edges labeled C (for Contain or Child). These edges form a tree on the nodes in the graph. The other edges in the graph represent either static dependencies between the parts of the software or permission.

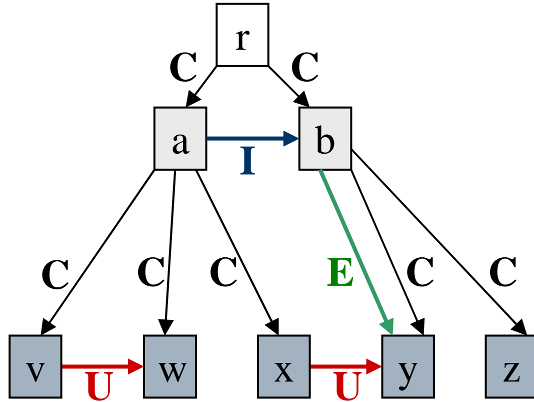

Figure 2. Example Nested Box and Arrow (NBA) graph.

NBA graphs have the following properties: They have no limit on their (finite) depth or size. In the hierarchy, there is no ordering among children of a node. Between two nodes, there can be more than one edge, but *only one* edge with any particular label.

The NBA model is related to the form of graphs used in the Rigi project [7], and is exactly the model used in University of Waterloo's SwagKit reverse engineering tool suite [9].

The following three definitions give a rigorous description of NBA graphs. We are giving a somewhat different, but equivalent, definition of NBA to that given by Malton [5] in order to simplify our presentation of permission theory. The first of these three defines relations, which are sets of edges.

Definition 1. Given a non-empty set of nodes N, a *relation* R is a possibly empty set of directed edges on N, i.e., $R \subseteq N \times N$.

For example, in Figure 2, the nodes N are {r,a,b,v,w,x,y,z} and relation U is {(v,w), (x,y)}

Definition 2. A non-empty set of nodes N, with a relation $T \subseteq N \times N$ is a (*directed*) *tree* if:

(1) *There is a single root.* That is, there is exactly one node r, the *root*, in N such that there is no edge (x,r) in T.

(2) *There are no cycles.* That is, there is no non-empty path of nodes $(x_1, x_2, \ldots, x_n)$ such that $x_1 = x_n$ and (x_i, x_{i+1}) is an edge in T.

(3) *Each node has at most one parent*. That is, for each node w in N, there is at most one edge (x,w) in T.

Definition 3. A *Nested Box and Arrow* (*NBA*) *graph* G = (T, s) consists of a tree and a state:

(1) *tree* T, which in turn defines a non-empty set of nodes N and a set of directed edges C, and

(2) *state* s, which is defined as a finite set of relations $\{V_1, V_2, \ldots\}$ on N.

The notation V_i stands for *variable* number i, and any edge in any Vi is called a *variable* edge. In Figure 2, the tree is given by relation C, and state s is defined by the three variable relations U, E and I. In our model of software architecture, variable relations represent dependencies and permissions.

We will consider that state s can be represented either as (1) a *set of named pairs* or as (2) a single *set of triples*, as follows:

(1) As NBA has been defined, state s consists of a set of labeled sets of pairs, e.g., in Figure 2, s consists of these three labeled sets of pairs:

U : {(v, w), (x, y)}

I : {(a, b)}

E : {(b,y)}

(2) When it is convenient, we will alternately consider that state s is represented as a set of triples of the form (x, V_i, y) where (x, y) is in relation V_i, e.g., in Figure 2, state s can be represented as this set of four triples:

{(v,U,w), (x,U,y), (a,I,b), (b,E,y)}

In an NBA graph, the tree can be defined by a set of pairs, e.g., in Figure 2, the set of pairs is:

C : {(r,a), (r,b), (a,v), (a,w), (a.x), (b,y), (b,z)}

Alternatively, we can consider that C is a set of triples, e.g., in Figure 2 this set of triples is:

{(r,C,a), (r,C,b), (a,C,v), (a,C,w), (a,C,x), (b,C,y), (b,C,z)}

3. RELATIONAL OPERATORS

Given labeled graphs such as NBA graphs, we can use Tarski's binary relational algebra [10] to query and manipulate architectural relations such as U and E [12, 11, 3]. To make our presentation self contained, we will briefly review such operators (see Figure 3), with examples based on Figure 2. Readers who are familiar with this notation can skip to the next section.

Operator Name	Example	Result
Union	$I \cup E$	{(a,b), (b,y)}
Subtraction	$C - E$	{(r,a), (r,b), (a,v), (a,w), (a,x), (b,z)}
Inverse	U^{-1}	{(w,v), (y,x)}
Subset	$E \subseteq C$	true
Composition	I o E	{(a, y)}
Identity	ID	{(r,r), (a,a), (b,b), (v,v), (w,w), (x,x), (y,y), (z,z)}
Transitive Closure	C^+	$C \cup$ {(r,v), (r,w), (r,x), (r,y), (r,z)}
Reflexive T.C.	C*	$C^+ \cup$ ID

Figure 3. Operators on Binary Relations (See Figure 2).

Ideally, the examples in Figure 3 make clear the meaning of the operators (for details see [3]), but for readers unfamiliar with this notation we explicitly define composition (the o operator):

Definition 4.

$V_1 \circ V_2 =_{def} \{(x,y) \mid \exists w \bullet (x,w) \in V_1 \wedge (w,y) \in V_2\}$

We will use these operators to define family relations in NBA graphs.

4. FAMILY RELATIONS

Given a hierarchy defined by tree relation C, we define derived relations, including *Parent* and *Ancestor,* as in Figure 4. We will use these *family* relations in our permission rules.

Family relation	Definition	Example edge
Self	ID	(b,b)
Parent	$P = C^{-1}$	(w,a)
Sibling	S = P o C - ID	(w,x)
Descendent	D = C+	(r,x)
Ancestor	A = P+	(x,r)
Super cousin	K = P* o S o C*	(v,b)

Figure 4. Family relations derived from tree relation C.

The K (*super cousin*) relation is the least obvious of the relations in Figure 4, so it will be described further. K edges connect every pair of nodes x and y such that x and y are distinct nodes and are neither descendents nor ancestors of each other. So K can alternatively be defined as

K = ALL – ID – A – D

where ALL = P* o C* contains all possible edges on node set N.

We will refer to family relations, including C and ID, as *constant* relations.

5. EXAMPLES OF PREMISSION RULES

This section presents four examples of rulesets. These determine which edges, such as U edges, are permitted in an NBA graph.

(1) Whole Import-Export ruleset. Our first example is presented in Figure 5.

> 1) A node may **export** its children.
>
> 2) A node may **import** its siblings. Also, it may **import** what its parent imports.
>
> 3) A node may **use** its siblings and what they export recursively. Also, it may **use** what its parent imports, as well as what they export recursively.

Figure 5. Whole Import-Export ruleset (informal).

This ruleset, consisting of three *rules*, is similar to scope rules in languages such as Euclid [4]. The ruleset is said to be *Whole*, as in Whole Import-Export, because only whole (not parts of) boxes are permitted to be imported and exported.

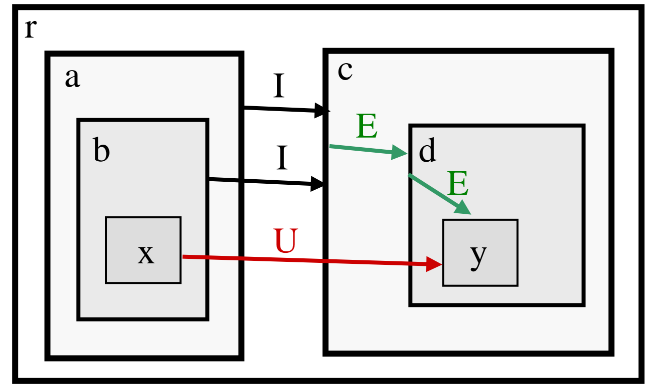

Figure 6. Illustrating Whole Import-Export ruleset.

We abbreviate *export* to E, *import* to I and *use* to U, and illustrate this ruleset using the NBA graph in Figure 6. In the figure, we represent the containment hierarchy by nesting of boxes, rather than by edges labeled as C, as was done in Figure 2. In the figure, edge (d,E,y) is legal by rule (1) because y is d's child. Edge (a,I,c) is legal by rule (2) because a and c are siblings. Edge (x,U,y) is legal by rule (3) because x's parent b imports c which recursively exports y. The rest of the variable edges in the figure can also be shown to be legal.

The Whole Import-Export ruleset is defined mathematically in Figure 7. In the notation used here, the composition operator o has higher precedence that $\cup$ (union), which has higher precedence than $\subseteq$ (subset). For example, the second rule in Figure 5 could be written as: $I \subseteq (S \cup (P \circ I))$. Transitive closure (*) has the highest precedence of all the operators.

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

Figure 7. Whole Import-Export ruleset, defined mathematically.

Mathematical specification of rulesets as in Figure 7 has the advantage of brevity and non-ambiguity over informal specification as in Figure 5.

(2) Selective Import-Export ruleset. We will now consider a second ruleset, given in Figure 8. Its third rule is identical to the third rule in the Whole-Import-Export ruleset (Figure 7). Its first and second rules are more permissive than the corresponding rules of the Whole ruleset, allowing descendants of (parts of) boxes to be imported or exported, when those descendents are transitively exported. Consequently, any NBA graph allowed by the Whole ruleset is also allowed by the Selective ruleset. For example, the graph in Figure 6 is allowed by the Whole Import-Export ruleset, so it is also allowed by Selective Import-Export. If we modified that graph by replacing edge (c,E,d) with edge (c,E,y), this would selectively export y from c and would disallow the use of d outside of c. The resulting graph would be allowed by the Selective but not by the Whole Import-Export rules. This sort of selectivity is particularly useful in graphs representing large software systems.

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

Figure 8. Selective Import-Export ruleset

(3) Tube ruleset. We will now consider a third ruleset, given in Figure 9. The Tube ruleset has only one rule, which is given on four lines in the figure. This ruleset is elegant in that it has only one kind of permission edge (T for tube). It has no explicit use (U) edges; instead it considers that T edges also represent *use* edges. So, for this ruleset, there is no difference between dependency edges and permission edges.

$T \subseteq S$
$\cup P \circ T$
$\cup T \circ C$
$\cup P \circ T \circ C$

Figure 9. Tube ruleset

If we temporarily take T to stand for "talk", the ruleset can be explained this way: a box can talk to its siblings, to what its parent can talk to, to the children of boxes it can talk to, and to children of boxes its parent can talk to.

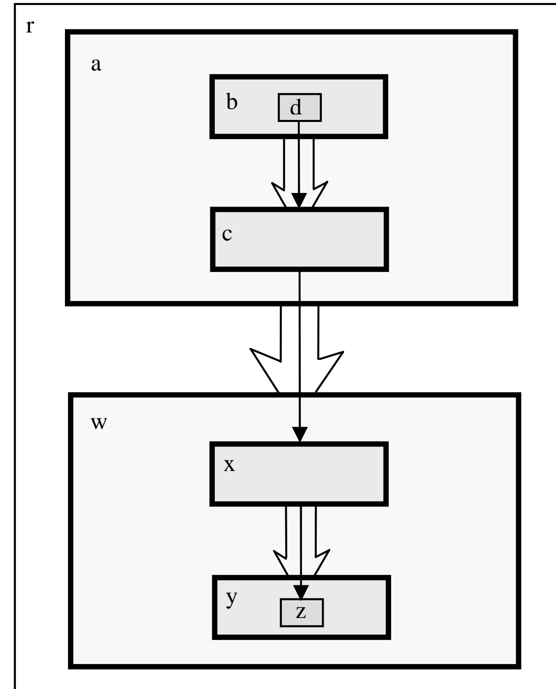

Figure 10. Illustrating Tube ruleset

Figure 10 illustrates a graph that is allowed by the Tube ruleset. In the figure, some of the edges are drawn as wide arrows, such as the arrow from a to w. This edge (a,w) is allowed because a and w are siblings (S) and similarly wide edges (b,c) and (x,y) are allowed as sibling edges. Edge (d,c) is allowed because d "is talking to what (c) its parent (b) can talk to". Edge (x,z) is legal because x "is talking to a child (z) of a box (y) it can talk to".

Figure 10 illustrates why this is called the *Tube* ruleset. Notice that wide edge (a,w) has a narrower edge (c,x) passing through it. In other words, (a,w) is a tube through which can passes edges such as (c,x). The general principle of this ruleset is that for an edge (a tube) to connect two boxes that are not siblings, it must tunnel through wider (higher level) tubes that provide higher level connections. For example, the tube (d,c) is allowed because it can tunnel through tube (b,c).

(4) Whole Buy-Sell ruleset. Figure 11 gives our fourth example of a permission ruleset. In the ruleset, E (export) is considered to mean *sell*, e.g., a parent box p can sell (export) its child x, thus making x available for use outside p. Symmetrically, parent p can specify that its child x can *buy* (B) boxes outside of p. While sell edges (E) are a mechanism for controlling access to boxes inside parent p, buy edges (B) are a mechanism for controlling access from items inside p to boxes outside of p. The third rule specifies that a use edge U can connect box x to box y only if x or its ancestor a is the sibling of y or y's ancestor b, where x reflexively transitively buys a and b reflexively transitively sells y. This ruleset is said to be *Whole*, as in Whole Buy-Sell, because only whole (not parts of) boxes can be bought and sold.

$E \subseteq C$
$B \subseteq P$

$U \subseteq B^* \text{ o } S \text{ o } E^*$

Figure 11. Whole Buy-Sell ruleset

Figure 12 illustrates a graph that is allowed by the Whole Buy-Sell ruleset. In the graph, E edge (y,z) is allowed by the first rule because y's child is z. This E edge allows boxes outside of y to use the z part of y. The B edge (x,w) is allowed by the second rule in Figure 12 because x's parent is w. This B edge allows one of w's parts, x, to use boxes that are outside of w. The U edge (x,z) is allowed by the third rule. The first rule is a mechanism to control information hiding, i.e., to determine what parts of an item are exposed to outside use. The converse of information hiding is implemented by the second rule, which provides a mechanism to control which internal parts have access to external items.

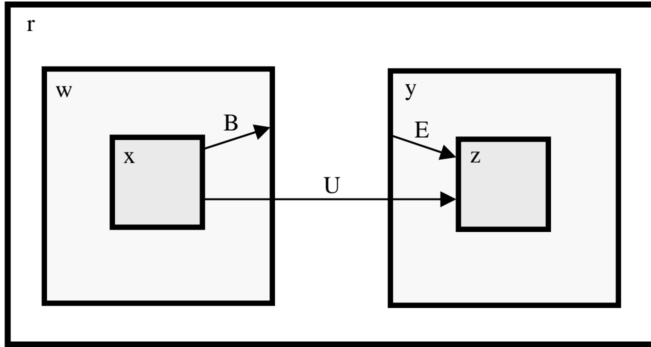

Figure 12. Whole Buy-Sell ruleset

The rulesets given in this section illustrate various kinds of permission that can used to control interactions in a software architecture. They also illustrate a common form for rules, which will be formalized in the next section.

6. RULESETS AS SUMS OF PRODUCTS

This section defines a particular form of ruleset, called Sum of Products (SoP), that we have illustrated in the examples in the preceding section.

In the example rulesets we have given, and more generally in every SoP ruleset, the right-hand side of each rule is a *sum* of *products*, e.g., in

$I \subseteq S \cup P \text{ o } I$

the right-hand side $S \cup P \text{ o } I$ is the sum (really, the union) of S and P o I.

P o I is the product (really, the composition) of P and I. S, as a single term, is considered to be a product.

In general in SoP rulesets, there is a rule for each variable, and the *left-hand side* of the rule consists of that variable. For example, for variable I, there is the rule $I \subseteq S \cup P \text{ o } I$, with I on the left-hand side.

The *right-hand side* is a union of compositions (possibly empty). The elements of each union can be variable relations such as I and E or constant relations such as C and P. Although we have illustrated transitive closure (*) on the right-hand side, to simplify the presentation we will not allow this in our definition of SoP rulesets. In the following sections, transitive closure could be allowed without changing the results.

We will now formalize what it means for state s in a given NBA graph to be legal with respect to a particular ruleset. State s *legal* if each of its variable relations satisfies its corresponding rule. For example, in the Whole Buy-Sell ruleset, E_s (relation E in state s) satisfies its rule if

$E_s \subseteq C$

In general, for any variable relation V, legality requires that V in state s (written as V_s) is a subset of its corresponding right-hand side in s, that is:

$V_s \subseteq f_V(s)$

where $f_V(s)$ is the function that maps state s to V's right-hand side value.

We will simplify this notation by assuming that s is represented as a set of triples, e.g., triples such as (a,I,b). We define permission function f such that f(s) maps s to the set of triples that are allowed by s by all the ruleset's right-hand sides. To illustrate, consider the example ruleset in Figure 13, which is the same the Whole Import-Export ruleset, but with the final U rule omitted. We call it the Simplified Whole Import-Export (SWIE) ruleset:

$I \subseteq S \cup P \text{ o } I$ $E \subseteq C$

Figure 13. Simplified Whole Import-Export (SWIE) ruleset.

For this ruleset, if s is the state in Figure 2 (but with U edges omitted), then function f, defined by the SWIE ruleset, maps s to this set of triples:

f(s) = {(a,I,b), (b,I,a),
(v,I,w), (v,I,x), (w,I,v), (w,I,x), (x,I,v), (x,I,w),
(y,I,z), (z,I,y),
(v,I,b), (w,I,b), (x,I,b),
(r,E,a), (r,E,b), (a,E,v), (a,E,w), (a,E,x),
(b,E,y), (b,E,z)}

The implication is that f(s) contains the triples that s permits. For a given tree T, f maps states to states, where each state consists of sets of triples. With this notation, we now formalize our definition of legality.

Definition 5. State s is *legal* according to function f, written $L_f(s)$, if each triple in s is contained in f(s). We write this as:

$L_f(s) =_{def} s \subseteq f(s)$

To summarize, in this section we have defined SoP rulesets, whose right-hand sides are unions of compositions, and we have defined what it means for a state to be legal for a given ruleset. In the next section, we will take a more abstract approach to permission and legality.

7. ABSTRACT PERMISSION THEORY

This section generalizes the definition of states to be independent of NBA graphs. We call this approach *Abstract Permission Theory* (*APT*),. Using this approach we will demonstrate that certain properties of rulesets are true without taking into consideration the structure of NBA graphs. This approach has the advantage of allowing us to concentrate on inherent properties of permission, independent of details about a particular graph model. Consequently, the results for APT can apply to permission systems other than SoP.

APT assumes that there is a finite set E of elements and that each state s is a subset of set E. As a special case, E could be the set of all possible variable edges in a tree in an NBA graph, as is the case for SoP. However, APT takes the abstract approach of considering that E is an arbitrary finite set. APT assumes that f is an arbitrary mapping from states to states, i.e., from subsets of E to subsets of E.

APT retains our previously introduced concept of legality, but now we will call it *f-legality* (because it is directly based on the permission function f). As before, we define:

$L_f(s) \; =_{def} \; s \subseteq f(s)$

We will introduce another concept of legality, called *π-legality,* and will explore the question of how these two kinds of legality are related. To define π-legality, we first define a *permission* relation π between states.

Definition 6. We define that state s *permits* state t, written s π t, as follows

$s \; \pi \; t \quad =_{def} \; s \subseteq t \subseteq f(s)$

This definition can be equivalently stated as follows. If s is f-legal ($s \subseteq f(s)$), and t is created by adding elements to s ($s \subseteq t$), and all elements added to t are f-legal according to s ($t \subseteq f(s)$) then s *permits* t.

If there exists state s that permits t (s π t), we say t is *permitted* and that t is π-legal, written $L_\pi(t)$:

Definition 7. $L_\pi(t) \; =_{def} \; \exists \, s \bullet s \; \pi \; t$

For example, suppose s is the state illustrated in Figure 2. Suppose that state t is the same as state s but additionally includes triple (w,U,y). Then, according to the Whole Import-Export ruleset, we can show that s permits t, i.e., s π t, and we can thus conclude that t is π-legal.

In the following, we will show that for every SoP ruleset, f-legality and π-legality are equivalent.

We will mainly be concerned with functions that are monotonric, defined as follows:

Definition 8. Function f is *monotonic* if for all s and t:

$s \subseteq t \quad \Rightarrow \quad f(s) \subseteq f(t)$

If f is defined by a SoP ruleset, it is necessarily monotonic. This follows from the observation that any function defined as a union of compositions is necessarily monotonic.

For a given state s, we define that an element is f-legal if it is contained in f(s). We can state monotonicity for SoP rulesets as follows: *Adding edges to a state can allow more edges to become f-legal, but can never cause existing f-legal edges to become illegal.*

We now consider the situation in APT in which state s is not necessarily based on an NBA graph and f is not necessarily based on SoP. We show that when f is monotonic, f-legality and π-legality are equivalent.

Theorem 1. If f is monotonic then for arbitrary state s

$L_f(s) \; = L_\pi(s)$

Proof. This will be proven by showing (1) that $L_f(s)$ implies $L_\pi(s)$ and (2) that $L_\pi(s)$ implies $L_f(s)$. (1) It is obvious that $L_f(s)$, i.e., $s \subseteq f(s)$, implies $L_\pi(s)$, i.e., $\exists \, t \bullet t \subseteq s \subseteq f(t)$, because t can be taken to be s. (2) To show the converse, that $L_\pi(s)$ implies $L_f(s)$, we must use monotonicity. We start with the assumption that $L_\pi(s)$ is true, i.e., that $\exists \, t \bullet t \subseteq s \subseteq f(t)$, and we conclude by monotonicity that $f(t) \subseteq f(s)$. (See Figure 14 for illustration.) Then, by transitivity of ⊆, we can conclude that $s \subseteq f(s)$, and hence that $L_f(s)$ is true. Therefore $L_\pi(s)$ implies $L_f(s)$. Having demonstrated both (1) and (2) we conclude that for monotonic f, $L_f(s) \; = L_\pi(s)$, as was to be proved. QED

If f is not monotonic, it is not in general true that $L_f(s) \; = L_\pi(s)$; that is, there exists f such that these two kinds of legality are not equivalent. However, for SoP rulesets, f-legality and π-legality are always equivalent.

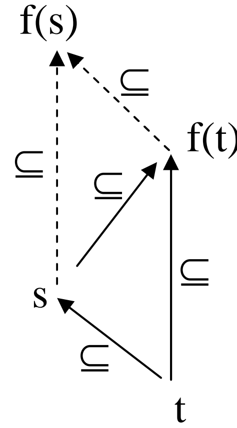

Figure 14. Showing $L_\pi(s)$ implies $L_f(s)$

Corollary. For any SoP ruleset, for any state s:

$L_f(s) \; = \; L_\pi(s)$

This follows from Theorem 1 and the observation that SoP rulesets are inherently monotonic.

Based on this theorem and corollary, when we are dealing with monotonic permission function f, or with SoP rulesets, we will simply use the term *legality*, rather than f-legality or π-legality, because the two concepts are equivalent.

8. CONSTRUCTIVE STATES AND PHANTOM STATES

This section defines what it means for a state to be constructive and for a state to be a phantom.

Informally, a state is *constructive* if it can be constructed step by step starting with the empty state φ (φ contains no elements of E), where each step adds legal elements to the state. Formally:

Definition 9. State s is *constructive* if $\phi \; \pi^* \, s$, that is:

$constr(s) \; =_{def} \; \phi \; \pi^* \, s$

In other words, when state s is constructive, there is a sequence of states $s_1, s_2, \ldots s_n$ where $s_1 = \phi$, $s_n = s$ and $s_i \, \pi \, s_{i+1}$.

This will be illustrated in terms of the Tube ruleset in Figure 15. In the figure, $\phi \, \pi \, s_1$, $s_1 \, \pi \, s_2$, $s_2 \, \pi \, s_3$ and $s_3 \, \pi \, s_4$. Hence $\phi \, \pi^* \, s_4$ and s_4

is constructive. The bottom graph in the figure represents the empty state ϕ, which is state s_1, for a tree consisting of nodes r, a, b, c, d and e. Using the first right-hand side (S) of the Tube ruleset (figure 9), we can construct state s_2 from s_1 by adding sibling edge (a,b), because s_1 permits s_2, i.e., $s_1 \pi s_2$. Then using the ruleset's second right-hand side (P o T), we can constructively add edge (d,b) to create state s_3. Then using the third right-hand side (T o C), we can construct state s_4. Therefore state s_4 is constructive, and so are states s_2 and s_3.

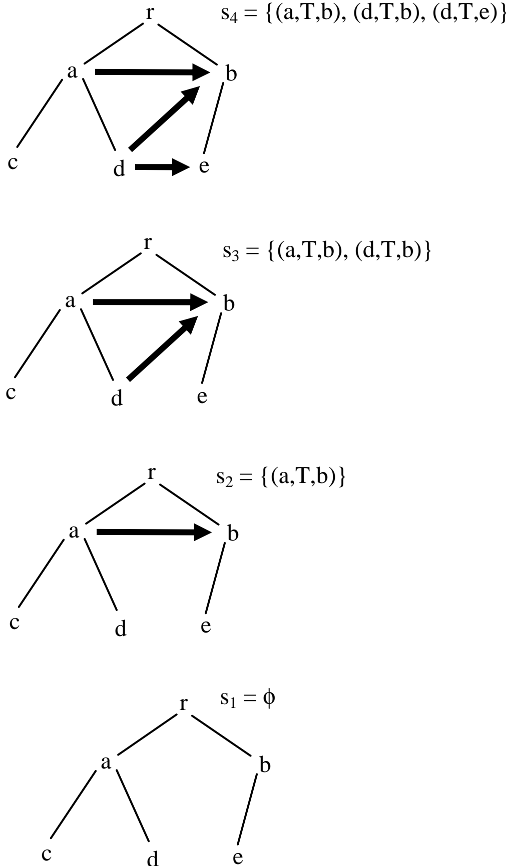

Figure 15. Showing s_4 is a constructive state

We state without proof that all example rulesets given up to this point in this paper have the property that for every state s, s is legal if and only if s is constructive.

It may come as a surprise that there exist SoP rulesets such that there are states that are legal but not constructive. Any such state is called a *phantom.*

Definition 10. State s is a *phantom* if s is legal but not constructive, that is:

phant(s) $=_{def}$ $L_f(s) \wedge$ not constr(s)

Since we apply these concepts to software architecture, we sometimes refer to phantoms as *phantom architectures.*

Figure 16 gives an example SoP ruleset that has a single rule. This ruleset is *phantomic,* i.e., it has legal states that cannot be constructed. This ruleset can be explained as follows. An E export edge can follow any C child edge, or it can follow the composition of two E edges. Although this seems to imply that every legal E edge must be a descendent edge, it turns out that this is not true, as phantoms (which are legal states) that contain non-descendent edges are possible.

$E \subseteq C \cup E \circ E$

Figure 16. A phantomic ruleset

As an example phantom, consider a tree containing node x, with a single variable edge (x,E,x), so state s is {(x,E,x)}. To determine if s is legal, we compute f(s), as follows.

f(s) = $C \cup E \circ E$

= $C \cup$ {(x,E,x) o {(x,E,x)}

= $C \cup$ {(x,E,x)}

Since s = {(x,E,x)} and f = $C \cup$ {(x,E,x)}, it follows that $s \subseteq$ f(s), and therefore s is legal.

We now show that s is not constructive as follows. Every constructive state must contain an E edge that is a descendent edge, because the only edges that can be constructed from ϕ are C edges, and these and successively constructed edges can only be constructed by composing them, via E o E, into further descendent edges. Since s contains no descendent edges, we conclude that s cannot be constructive. Since s is legal but not constructive, it is a phantom. Besides showing that s is a phantom, we have answered affirmatively the more general question: *Do there exist SoP rulesets that allow phantoms?*

9. PHANTOMS AND FIXPOINTS

Up to this point, this paper has explored permission theory and the concept of phantoms as motivated by considerations in software architecture. This section explores a related but different question: Are there areas of mathematics that effectively model the results we have presented? We will answer this question by showing that the theory of fixpoints, as somewhat expanded, provides a necessary and sufficient condition for monotonic function f to have phantoms. Throughout this section, we will assume that f is monotonic.

Given a function f, a fixpoint (or fixed point) s is a state such that f maps s to itself. If f maps s to a superset of s, we say s is a prefixpoint. More formally:

Definition 11. State s is a *fixpoint* (*fp*) when s = f(s). State s is a *prefixpoint* (*pfp*) when $s \subseteq f(s)$, that is:

fp(s) $=_{def}$ s = f(s)

pfp(s) $=_{def}$ $s \subseteq f(s)$

Some authors use the term postfixpoint instead of prefixpoint.

By definition, s is f-legal when $s \subseteq f(s)$ and hence s is a prefixpoint if and only if s is f-legal. If follows from our preceding theorem that for monotonic f, s is a prefixpoint if and only if s is π-legal. Note that for monotonic f, empty state ϕ is inherently a legal state and a prefixpoint.

Given that f is monotonic, and based on the subset ordering relation $\subseteq$ the classical Tarski-Knaster theorem applies:

Tarski-Knaster Theorem. $f^{\omega}(s)$ is a fixpoint. It is a least fixpoint.

So, if f is applied to empty state ϕ, then applied to the result of that, then to the result of that, etc., eventually we find a fixpoint state $s = f^{\omega}(f)$, such that

$$f(s) = s$$

State $s = f^{\omega}(s)$ is a least fixpoint, which means that there is no t, $t \subset s$, such that $f(t) = t$.

Figure 17 illustrates a state space with empty state ϕ on the bottom and a fixed point (fp) on the top. The states in the outlined area are prefixpoints; they are the legal states. We can think of constructive chains leading from legal states at lower levels in the diagram to legal states which they transitively permit at higher levels.

The diagram labels ϕ and two other points as spfp (strong prefixpoint). This concept will now be defined.

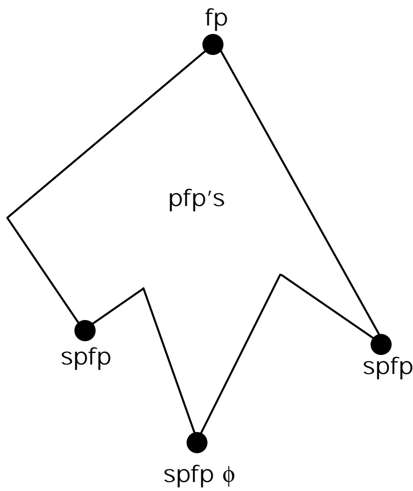

Figure 17. Illustration of fixpoints.

Definition 12. State s is called a *strong prefixpoint* (*spfp*) if s is a prefixpoint and there is no state t less than s that permits s, in other words:

$$spfp(s) =_{def} pfp(s) \wedge \text{not } \exists\, t \subset s \bullet t \text{ pi } s$$

A strong prefixpoint is a kind of minimum, i.e., it is a point that is permitted (by itself), but is permitted by no state beneath it. This is illustrated in Figure 17 by the fact that the three strong prefixpoints have no legal states below them. It is clear that ϕ is an spfp state because it is a legal state and there is no state smaller than it. Any non-ϕ spfp state is phantom, as is shown in the following lemma.

The definitions of fp and pfp are from the literature, but the definition of spfp is new in this paper.

Figure 17 shows a single fixpoint (fp), which is appropriate if function f has a single fp. If f has more than one fp, the diagram should be extended to contain a corresponding number of outlined areas, each with one fp on the top and at least one spfp on the bottom.

Lemma 1. If s is a non-ϕ strong prefixpoint, then it is an phantom, that is:

$$spfp(s) \wedge s \neq \phi \;\Rightarrow\; phant(s)$$

Proof. Recall that s is defined to be a phantom if it is legal but not constructive. If s is a strong prefixpoint, then it is necessarily legal. So we need to show only that s, a strong prefixpoint, is not constructive. To do this, we need to show that there is no chain $\phi\ \pi\ s_1\ \pi\ s_2 \ldots s_n\ \pi\ s$, that would construct s. Since s is non-ϕ and since s is spfp, we know there is no state less than s that permits s, so a chain leading from ϕ to s, cannot exist. Consequently, s must be a phantom. QED

Based on this lemma, we can consider that each strong prefixpoint is a minimal phantom, i.e., a phantom that is allowed by no state less than itself. A system contains such minimum phantoms exactly when it contains any phantoms, as will now be shown.

Theorem 2. Monotonic function f is phantomic (allows phantoms) if and only if f has one or more non-ϕ strong prefixpoints.

Proof. We will take LHS to be "f allows phantoms" and RHS to be "f has one or more non-ϕ strong fixpoints". We will prove the theorem by proving (1) RHS $\Rightarrow$ LHS and that (2) (not RHS) $\Rightarrow$ (not LHS). To prove part (1), we use the preceding lemma as follows. Since f has a non-ϕ spfp state, it follows that s is a phantom, so RHS $\Rightarrow$ LHS. Proceeding to part (2), we will now prove that (not RHS) $\Rightarrow$ (not LHS). Suppose that RHS is false, i.e., that there is no non-ϕ spfp state. This implies that for every legal state u, there exists state t, $t \subset u$, such that $t\ \pi\ u$, which implies there must exist a chain $\phi\ \pi\ s_1\ \pi\ s_2 \ldots s_n\ \pi\ s$ and thus that s is be a phantom. Hence no states can be phantoms. QED

This theorem characterizes, in terms of fixpoints, when monotonic function f allows phantoms. As such, it links permission theory for software architecture to the theory of fixpoints. It might appear that we can conclude from this theorem that we can in general determine if a given function f allows phantoms, but this does not follow from the theorem. Indeed, as the following section concludes, the question of whether it is decidable if a Sum of Products ruleset is phantomic remains open.

10. SIGNIFICANCE OF PHANTOMS

Phantoms arise out of the formalization of permission theory. They are an anomaly that can be considered analogous to anomalies such ambiguity in context free grammars or imaginary numbers in number theory.

We can choose to ignore phantoms as a bothersome mathematical artifact. However, it seems interesting to gain some understanding of them for various reasons. First, phantoms are intellectually interesting as a "surprise" in a mathematical formulation. Next, it may be that a designer of a ruleset does not realize that phantoms are possible, and so might wrongly conclude that legality and constructivity are identical concepts. As a result, any unsuspected phantoms would violate the *no surprise* rule and could lead to design errors.

Phantoms can be dealt with in various ways. If the SoP ruleset of interest does not allow phantoms, then things are simple and we can use the same test, $s \subseteq f(s)$, to determine both legality and constructivity as the two concepts are identical in this case. If the ruleset allows phantoms, we might require all legal states to be constructive, effectively defining *strong* legality, which requires both (ordinary) legality and constructivity.

Ideally, we should execute an algorithm that reads an SoP ruleset and determines if phantoms are possible for that ruleset. The author has developed a set of conditions that can be imposed on SoP rulesets to check for these conditions. The check can be executed to eliminate rulesets that allow phantoms. (There is insufficient space to give these conditions in this paper.) However, the author has not been able to discover an algorithm that decides if arbitrary SoP ruleset is phantomic. So we pose this challenge to the reader:

Open Question. *Is it decidable if arbitrary SoP ruleset allows phantoms?*

Perhaps a reader of this article can answer this question. The author conjectures that this is decidable.

11. PRACTICALITY AND RELATED WORK

In the history of programming methods, from structured programming to object-oriented programming and beyond, we have seen the development of practical mechanisms that help us understand and control interactions in software. Control of interactions in large software systems is essential in the maintenance and understanding of their architectures.

This paper presents and analyzes the properties of a mechanism, SoP rulesets, that can be used for that purpose. SoP rules are easy to implement: they can be enforced by a few lines of code written in a language such as Grok that supports binary algebra operators [3].

They are efficient, in that the check to see if a graph is legal can be executed in a few minutes or seconds even for models of target software systems whose source code exceeds a million lines of code. With appropriate use of hashing and radix sorting, these algorithms effectively run in time O(E) where E is the number of edges in the model. What is less obvious and needs further study is whether such mechanisms to control dependencies can be successfully used in the ongoing maintenance of large software systems [12].

Access rules in module interconnection languages (MILs) are closely related to the concept of permission presented in the present paper [2].

Mancoridis [6], as the author's PhD student, developed a visual notation to specify permission which is related to SoP rulesets. Mancoridis' notation is non-monotonic in that it includes "negative" permissions; adding such an edge can cause existing legal edges to become illegal. Lacking monotonicity, a result like the one given in this paper in which f-legality is equivalent to π-legality does not hold.

Most of the example rulesets given in the present paper appear in a somewhat different form in a 1996 technical report by the author [3] and in Mancoridis' PhD thesis. That technical report also introduces SoP rulesets.

The author knows of no work that recognizes phantoms or generalizes permission to the level of Abstract Permission Theory.

12. CONCLUSIONS

This paper has presented a graph model, NBA graphs, of the structure of software architecture. This model is hierarchical and can include dependency edges between nodes. We showed how permission edges can be used to control dependencies. After giving examples of rules, we formalized a class of permission rulesets called Sum of Products.

The concept of permissions in the graph models was generalized to Abstract Permission Theory in order to separate details about the particular model from deeper concepts.

It was shown that two concepts of legality (based on permission function f or on permission relation π) are equivalent for monotonic permission rules such as SoP.

There are *phantom* graphs that satisfy rulesets but cannot be constructed. We left as an open question whether there is an algorithm to decide if an arbitrary SoP ruleset allows phantoms.

Control of dependencies is an essential part of the design of software architecture, but we are still lacking good mechanisms that scale up to handle this problem for large systems. It is hoped that the mechanisms and analysis given in this paper will help guide the way toward developing such mechanisms.

13. ACKNOWLEDGMENTS

Discussions with Vassilios Tzerpos provided essential intuition about SoP systems.

14. REFERENCES

[1] I. T. Bowman, R. C. Holt, N. V. Brewster. Linux as a case study: Its extracted software architecture. In *ICSE '99: International Conference on Software Engineering*. Los Angeles, 1999.

[2] F. Deremer, H. H. Kron. Programming-in-the-large versus programming-in-the-Small, IEEE Transactions on Software Engineering, Vol SE-2, No 2, June 1976

[3] Richard C. Holt. Binary Relational Algebra Applied to Software Architecture, CSRI Technical Report 345, Computer Systems Research Institute, University of Toronto, June 1996.

[4] B. W. Lampson, J. J. Horning, J. G. Mitchell, G. J. Popek. Report on the Programming Language Euclid, ACM SIGPLAN Notices 12, 2 (February 1977), 1-19.

[5] A. J. Malton, Holt, R.C. Boxology of NBA and TA: A Basis for Understanding Software Architecture, WCRE 2005: Working Conference on Reverse Engineering, Nov. 2005.

[6] S. Mancoridis. Controlling the Interactions of Architectural Design Components using Scoping Rules, PhD Thesis, Department of Computer Science, University of Toronto, Dec 1995.

[7] H. A. Müller, S.R. Tilley, K. Wong. Understanding software systems using reverse engineering technology: perspectives from the Rigi project. In *Proceedings of CASCON '93*, IBM, 1993.

[8] M. Storey, CHiSEL Group (University of Victoria). The SHriMP application and technique. See http://www.thechiselgroup.org/ shrimp

[9] SWAG: Software Architecture Group (University of Waterloo). The lsdit graph visualization tool. See http://swag.uwaterloo.ca/lsedit/.

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

[11] Arie van Deursen, Leon Moonen. An empirical study into COBOL type inferencing, Science of Computer Programming (Elsevier), 40 (2001) 189-211.

[12] Rob van Ommering, René Krikhaar, Loe Feijs. *Languages for Formalizing, Visualizing and Verifying Software Architecture*, Computer Languages 27 (2001), p3-18.