Grokking Software Architecture

Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo, Canada
holt@uwaterloo.ca

Abstract

This paper is a retrospective arising from a WCRE paper published in 1998 promoting a relational approach to manipulate software architecture and to help solve various problems in software analysis. That paper explains how Tarski's binary relational algebra, embedded in a language such as Grok, can solve such problems. Tarski's notation is elegant and often efficiently executable when the subject at hand is characterized by parts with binary relations between them. Software architecture, especially as-built architecture, is such a subject. This paper concentrates on the following three questions. What impact can a relational approach have on our understanding of software architecture? What Grok-languages exist and what are their strengths? How have Grok-like languages been used to solve problems in software architecture or software analysis?

1. Introduction

The word *grok* was coined by Heinlein in 1961 in his science fiction novel *Stranger in a Strange Land*. This word has come to mean "to share the same reality or line of thinking with another physical or conceptual entity."[39] The Grok language discussed in the present paper was so named because one of its purposes is to help people to think about and to share understanding about software architecture [20].

The Software Bookshelf project [12] which had as a goal to reverse engineer large commercial software, such as IBM's code optimizer TOBEY, was a springboard for a various concepts about software architecture. One spin-off of this project was the Grok language.

The Grok language [17] is based on Tarski's algebra [34] for binary relations. The language and its use as a tool to be applied to software architecture was described in a 1996 technical report [17] and then in a 1998 paper [17] that appeared in WCRE (Working Conference on Reverse Engineering). In 2008 (now), the 1998 paper was selected as the most influential paper appearing in WCRE 1998. The current paper is a retrospective written at the invitation of WCRE about that 1998 paper. The title of the present paper, Grokking Software Architecture, is an intended *double entendre* meaning both "Understanding Software Architecture" and "Using the Grok Language to Manipulate Software Architecture."

The novelty of the 1998 paper was to propose the following: Various manipulations that are done to architectural structures to make them comprehensible can be neatly specified and automatically implemented based on Tarski's notation. Along with a 1996 technical report, various architectural uses of this approach, such as enforcement of architectural constraints, were suggested. The approach also promoted the use of typed graphs (graphs with various types of edges) as a simple and convenient mathematical basis for representing architectural structures.

This paper is organized around the following three questions:

1) ***What impact can a relational approach have on our understanding of software architecture?*** To answer this question, Sections 2 and 3 concentrate on two views of architecture: the logical and as-built views. Sections 4, 5 and 6 discuss boxology and typed graphs along with Tarski's relational algebra, with an emphasis on how a relational approach provides appropriate underpinnings for understanding and reverse engineering of software architecture.
2) ***What Grok-like languages exist and what are their strengths?*** Section 7 gives the history of Grok-like languages, while Section 8 discusses the implementation and performance of these languages.
3) ***How have Grok-like languages been used to solve problems in software architecture and software analysis?*** Section 9 gives examples of use of Grok-like languages to solve architecture and software analysis problems

We now begin with the first of these questions.

2. Software architecture: What is it? What good is it?

To address the question of how a relational approach should impact software architecture, this section and the next concentrate on software architecture and views of architecture.

What is software architecture? There are many definitions of software architecture; see the Software Engineering Institute's list of these [31]. There is even an ANSI/IEEE [1] standard for software architecture documentation. Still there is far from universal acceptance of a definition for this term. As Gorton [15] says, "I have a sneaking suspicion that 'architecture' is one of the most overused and least understood terms in professional software development circles."

There are two central concepts that appear in most of these definitions: (1) division of the target software system into system-level parts (components, modules, files, classes, etc.) and (2) interconnections between these parts (communication paths, data flow, procedure/method calls, includes, imports, references to variables/types, etc.) As Gorton [15] succinctly puts it, "Architecture captures system structure in terms of components [parts] and how they interact." This suggests, not surprisingly, that a graph formalism that models parts (as nodes) and interactions (as edges) is well suited to model the structure of software architecture.

In much of the writing about software architecture, there is an underlying theme that things would be better if software systems had more architectural documentation. Architectural Description Languages (ADLs) were created to support this theme, but somehow they have not been up to the job. Mary Shaw [33] says, "UML has, for better or (many would say) worse, become the industry standard ADL [Architecture Description Language]," and she says, "It still lacks, however, a robust suite of tools for analysis, consistency checking, or other means of automatically connecting the information expressed in UML with the system's code." So, after years of effort to invent appropriate ADLs, we have nothing better than UML's box-and-arrow diagrams. UMLs diagrams that apply at the architecture level are not bad, but are not much improvement on the box-and-arrow diagrams that have been in use for many years.

What does this tell us? My feeling is that box-and-arrow diagrams, formalized as typed graphs and manipulated by Grok-like languages go some distance toward providing the basis for the "robust suite of tools" that Shaw would like to see.

Some writers believe that most software systems have messy architectures. As Foote [13] puts it (perhaps tongue in cheek): "The architecture that actually predominates in practice is the 'big ball of mud'," also known as spaghetti code, shanty town, can of worms, dog's breakfast, etc. Perhaps Foote is being too pessimistic. My observation is that in many cases, the architecture of a software system is reasonable, although its documentation is lacking. As Gorton [15] observes, "It's common for there to be little or no documentation covering the architecture in many projects."

If documentation for architecture is so important, why is it so often lacking? We can get a hint about the answer to this question from super-programmer Linus Torvalds [35] who says, "I'm hopeless when it comes to documentation." Perhaps he is reflecting the simple truth that spending time writing good code is usually much more important than spending that same time documenting architecture. Still, in most systems of any size, such as Torvald's Linux, I expect that there is a fairly coherent architectural design in the head of the chief architect. (I use the word *wetware* to describe such architectural knowledge in a person's brain.)

In those developments in which there is time set aside for documenting architecture, how is this documentation actually created? Gorton [15] says that he is "pretty certain the predominant tools used for architecture documentation are Microsoft Word, Visio and Power Point, along with their non-Microsoft equivalents." These tools that Gorton lists are easy to use to produce documentation that is reasonably easy to update manually. Visio provides a handy, brute-force way of producing box-and-arrow diagrams. Still, these tools are a long way from providing Shaw's hoped for suite of tools. An essential disappointment is that they do not generally incorporate the extensive amount of information that can be automatically extracted from the target software system. Also, they provide no means other than manual manipulation to keep the box-and-arrow diagrams up to date.

From these observations, I conclude that what is needed are concepts, notations and tools that are easy to learn and use and which help us produce useful, understandable documentation, produced with minimal effort.

From this comes a rule of mine about software architecture and its documentation: "Only as much architecture as is needed." Over enthusiastic efforts to make the architecture too beautiful or to create too much documentation detracts from the project's central goals (short time to market, new features, more testing, etc.) Beware of gilding the lily. My position in this

paper is that the best way to improve software architecture lies in lightweight, easy to produce, understand and maintain methods and tools for documenting architecture. Briefly put, we need agile architecture documentation tools. As will be presented in more detail below, I suggest that these concepts and tools can be based on typed graphs with Grok-like languages to manipulate them.

3. As-built architecture

There are many ways of viewing software architecture. The parable of the elephant and the blind men illustrates these views, told here from the Jain perspective [21].

Once upon a time, there lived six blind men in a village. One day the villagers told them, "Hey, there is an elephant in the village today." They had no idea what an elephant is. They decided, "Even though we would not be able to see it, let us go and feel it anyway." All of them went where the elephant was. Everyone of them touched the elephant. "Hey, the elephant is a pillar," said the first man who touched his leg. "Oh, no! it is like a rope," said the second man who touched the tail. "Oh, no! it is like a thick branch of a tree," said the third man who touched the trunk of the elephant. "It is like a big hand fan" said the fourth man who touched the ear of the elephant. "It is like a huge wall," said the fifth man who touched the belly of the elephant. "It is like a solid pipe," Said the sixth man who touched the tusk of the elephant. They began to argue about the elephant and everyone of them insisted that he was right. It looked like they were getting agitated. A wise man was passing by and he saw this. He stopped and asked them, "What is the matter?" They said, "We cannot agree to what the elephant is like." Each one of them told what he thought the elephant was like. The wise man calmly explained to them, "All of you are right. The reason every one of you is telling it differently because each one of you touched the different part of the elephant.

Kruchten's [25] seminal paper lists four views of a software architecture, which can be thought of as the insights from four blind men: (1) *logical* (end-user's view of functionality), (2) *as-built* (developer's view of the implementation), (3) *process & performance* (integrators view) and (4) *physical* (system engineer's view of communications).

My feeling is that the first of Kruchten's views, the logical view, is fundamental because it tells the purpose of the system. It explains what the system does. That is important to almost all stakeholders. I feel that the second of these, the as-built view, comes next in importance --- because the system cannot exist without its underlying implementation and because the code is in a sense the ultimate specification of the actual system. As programmers sometimes say, "code is king."

(The names of and the details of these views vary from author to author. Kruchten calls the as-built view the *development* view and I usually call it the *concrete* view. I usually call the logical view the *conceptual* view.)

From the perspective of the reverse engineer, the as-built view is fundamentally important, because the engineer works directly with the code (often the source, sometimes only the binary) and seeks to create, with limited resources and limited time, a useful view of the software system. From Chikofsky's [7] classical definition, "Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction." Commonly some sort of parsing of the code is done and *facts* are collected about parts of the code, for example, about how the code is stored in files and about how parts of the system interact. The interactions that are easiest to extract are static dependencies, such as whether procedure P calls procedure Q. These facts are commonly represented as triples, such as (Call, P, Q), which can be interpreted as edges in a typed graph.

Grok-like languages as well as relational systems such as SQL can read, query, abstract and manipulate the typed graphs that consist of these edges. As well, there are many tools for visualizing these graphs. So, not only are as-built views important, they are also the easiest to produce from that all important artifact, the source code. They are also easy to keep up to date, because they can be (at least in principle) re-extracted in a largely mechanical way, from the source code as that code changes.

In brief, there are many tools to help create as-built views, starting with the source code, so the cost of creating such views is small. So, the as-built view gives us lots of "bang for the buck", i.e., useful architecture documentation at modest cost.

By comparison, creating and maintaining detailed documentation of the logic view is expensive, slow and largely manual. Meantime, the wetware documentation inside the architect's head generally includes a logical view. Since that view already exists, as wetware, there is not such a need to reproduce it as actual documentation. (Still, it must be remembered, if the architect departs from the project, the wetware also departs.)

I conclude this section by observing that box-and-arrow diagrams are a natural fit for modeling the structure of software architecture, especially the as-built view. Taking this as a clue, the next sections discuss fruitful ways to formalize and mechanize box-

and-arrow diagrams, with a goal of making their creation, manipulation, visualization and grokking easier and more efficient.

4. Boxology

Wikipedia [40] says, "A *boxology* is a representation of an organized structure as a graph of labeled nodes ('boxes') and connections between them (as lines or arrows). This concept is useful because many problems in software design are reducible to modular 'black boxes' and connections or flow channels between them. The term is somewhat tongue-in-cheek and refers to the generic nature of diagrams containing labeled nodes and (sometimes directed) paths between them."

Wikipedia's presentation characterizes the nature of much of the structure in software architecture, and helps explain why architecture so often uses box-and-arrow diagrams. As Gorton [15] observes, "the most widely used design notation is informal 'block and arrow' diagrams."

Shaw [32] wrote an article titled, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems." This title reflects the ubiquitous use of box-and-arrow architecture diagrams, with the humorous implicit reference to Roger Tory Peterson's [30] *Field Guides* for identifying birds, animals, etc.

There is discontent about boxology, for example Mitra [29] states, "The box-and-line diagram of the architecture leaves a lot of room for interpretation." He is right, but the needed interpretation can be added without spoiling the simplicity and elegance of these diagrams. It can be added manually, for example, by accompanying written documentation created using an editor such as Microsoft Word or more formally using attribute values attached to nodes and edges (to boxes and arrows).

Some people might argue that it a show-stopper that box-and-arrow diagrams do not inherently contain the semantics needed to describe their intended meaning. I argue the opposite: That this is an advantage, rather than a disadvantage, in that this allows us a separation of concerns in which we can master and automate box-and-arrow manipulation without being bogged down with the complexity of the semantics. Consider a similar case, namely Context Free Grammars. Context Free Grammars are powerful and useful to formalize syntactic aspects of programming languages. By themselves they do not provide the language's semantics. CFGs and box-and-arrow diagrams (formalized to typed graphs) are powerful and useful because they provide a convenient representation for structures to which semantics can readily be attached.

5. Typed graphs

This section and the next one will dive to a deeper level of precision, discussing how to formalize box-and-arrow diagrams as typed graphs [27]. This formalization allows us to be mathematically clear about the structure of these diagrams. In this formalization, we use graphs that consist of nodes (for boxes) and directed edges (for arrows). We will use typed graphs, in which there is more than one type (or color) of edge. For example, there might be CALL edges representing calls from one method to another and INHERIT edges representing the inheritance hierarchy. In the Grok language, as well as languages such as Relview [2] and CrocoPat [3], there are distinct edge types R_1, R_2, ... R_n. In these languages, nodes do not have types. However, by convention a node can be tagged to have a type by the convention that self-loops of type T_x can mark nodes of type x. Grok allows *attributes* which are named string values to be attached to nodes and to edges. These attributes can be interpreted to give meaning to the graph.

It is convenient in architectural diagrams to have nested boxes, for example, to represent classes in packages or methods in classes. Feijs [11] formalizes nesting in RPA (Relation Partition Algebra) by augmenting typed graphs with a partitioning of the nodes. Each set of partitioned nodes is considered to be nested in its partition. RPA leads to elegant mathematical characterization of concepts such as *lifting* (aggregating connectivity within a partition). Grok represents nesting in a different way, by designating one particular edge type *contain* to represent nesting. It is assumed that *contain* defines a tree across the nodes of the graph. Grok's approach has various advantages. It avoids complicating the formalism with the concept of partitions (although these can be defined in terms of the *contain* tree), and it supports multiple levels of nesting, by means of each level of depth of the tree. With this convention, a language such as CrocoPat, which does not explicitly support hierarchies, can still effectively handle nesting by designating one relation as the *contain* relation.

Given typed graphs, it is important to have some way, ideally an algebra, to manipulate them.

6. Tarski binary relational algebra

This section explains how Tarski binary relational algebra can raise the level of abstract of typed graphs

so they can be manipulated independently of data structures used to implement or represent them.

We will consider that the data representing an architectural structure is a typed graph. If we define operations, such as Tarski operators, that manipulate that graph, then we have defined a *data model*. Such a model implies that we can manipulate the graph without knowing about its underlying encoding or implementation. Instead we manipulate the graph using high level operators. This abstraction of graphs up to a higher level is similar to the abstraction that occurs when the data of a database is accessed only by high level operations such as SQL commands. This high-level access implies that using databases (and typed graphs) is greatly simplified, hiding data structures such as linked lists, allowing operators to be optimized separately from their applied usage and providing high level operations at the level of joins and unions.

Tarski algebra allows a relation R_1 (a set of edges with a given type) to be combined with another relation R_2 to produce their union, intersection, difference, composition, etc. The Grok language adds various operations such as transitive closure along with many housekeeping features such as file management, loops and *if* statements, etc.

My 1998 WCRE paper suggested that a language such as Grok with embedded Tarski operators is an elegant and efficient way to query and manipulate architectural graphs. At almost the same time or a bit later, Feijs et al. [11] made similar suggestions and then continued to expand upon these ideas over the last decade.

In data base theory there is the concept of expressiveness of queries. Expressiveness determines what information a query can extract from a data base. It is known that Codd's n-ary relational algebra is somewhat more expressive than Tarski's binary algebra. The basic reason is that Tarski's queries are limited to producing sets of pairs, while Codd's can produce sets of n-ary tuples.

Note: Codd's algebra can be applied to a typed graph, because such a graph is essentially a set of 2-ary tables but Tarski's algebra can be applied to n-ary tables only when n is limited to 2.

It was not obvious in 1998, but became increasingly clear with the years, that expressive power is significant in the case of querying architectural diagrams [10]. As will be seen below, Beyer [4] has developed an a query language CrocoPat which can be applied to typed graphs to produce n-ary tables, thus giving it more expressive power than Grok.

7. History and comparison of Grok-like languages

This section and the next consider the second question asked at the beginning of this paper: *What Grok-languages exist and what are their strengths?*

For the purposes of this paper, a language is *Grok-like* if it can manipulate typed graphs using high level operators, ideally with a precise mathematical definition of those operations. Such languages can be used in an application-independent way to do various kinds of analysis and manipulation of typed graphs, including graphs that come from disciplines other than program analysis. However, to be Grok-like, these languages should implemented such that they are useful for program analysis.

Table 1 gives a list of such languages, which we will discuss.

The mother of all Grok-like languages is Prolog. I used it on architecture graphs, as early as about 1995. At that time, with the Prolog implementations I tried, it did not scale to handle the many thousands of edges in my architectural diagrams, so I abandoned it. This lead eventually to the development of Grok. Beyer [4] gives more recent experience of using Prolog for architectural purposes, reporting that it is "prohibitively inefficient for practical software analyses of this type."

Language	Author	Date
Prolog	Colmerauer et al.	1972
SQL	Chamberlin & Boyce	1974
GraphLog	Consens et al.	1989
Relview	Berghammer et al.	1993
Grok	Holt	1996
RPA	Feijs et al.	1998
GReQL	Kullbach & Winter	1999
JGrok	Wu	2001
CrocoPat	Beyer	2003

Table 1. Grok-like languages

SQL, as well as Datalog, also does not perform well (it is too slow) for work on architectural graphs, so it is not really a Grok-like language [3]. There have been many instances of embedding SQL in another language such as C. We are ignoring these embeddings as they are not known to have been used for efficient architectural analysis.

Consens [8] and colleagues [28] developed and demonstrated GraphLog, which supports high level graphical queries. They described how the notation can be used to compute metrics, impose constraints and locate instances of design patterns. In a sense, GraphLog is not a Grok-like language in that it does

not have a script form which is often appropriate for architectural manipulation.

Relview [2] is the first computer language to my knowledge to be based on Tarski algebra. It is apparently the only language other than Turing to be so based. While it predated my work on Grok by about 3 years, I was not aware of it until I had Grok working on large architectural diagrams at IBM's Toronto lab. I tried running a version of Relview, but at the time concluded that it was not robust enough for the architectural work I was doing. My feeling was that Relview was excellent for teaching and experimentation with Tarski algebra but was not suited for program analysis. Beyer [4] has tested Relview more recently with reasonable results.

Soon after I developed and used early versions of Grok, a group of Dutch researchers (Feijs, Krikhaar [11] et al.) began applying Tarksi algebra to architectural problems. That group proceeded to experiment with, to advance and to publish the state of the art of using Tarski algebra, enhanced to Relation Partition Algebra, for architecture analysis. That group did not, to my knowledge produce an actual language to support the algebra, but rather ran command scripts to invoke individual programs that carried out Tarski operations.

About that time, but perhaps a little later, a German group (Kullback, Winter, Ebert et al.) developed a notation and computer support for GReQL [24]. GReQL looks much like SQL but is used for querying typed graphs. It can extract n-ary patterns. As opposed to Grok, GReQL does not update graphs, so it was more designed to recognize patterns rather than to transform architectural structure.

In 2001, Jingwei Wu, a graduate student of mine, re-implemented Grok, the result being JGrok. JGrok is similar to Grok, but its syntax looks more like Java, in which it is written, whereas Grok looks more like Turing, in which it is written. JGrok supports n-ary values and operators, of a limited variety. It can be easily combined with other tools due to the fact that it is written in Java. JGrok's internal data structures are based on linked lists, which are completely different from Grok's data structures (described below).

At about that time or a bit later, Beyer [3] developed a Grok-like language called CrocoPat. See Beyer's [4] comparison of CrocoPat to Grok.

CrocoPat's notation (called RML) uses free variables (typically called x, y, z, ...) that allow it to deal with n-ary relations. (Note that free variables are sometimes called *attributes*.) This allows CrocoPat to extract n-ary patterns which Grok is not able to do. RML can be thought of as a variant of Datalog.

By contrast with CrocoPat, Grok does not use free variables, because its restriction to binary relations makes these unnecessary. For example, the Grok statement

```
S := P ○ C
```

computes the composition of the parent relation P with the child relation C to produce the reflexive sibling relation S. In other words, S is the composition of P and C. Written in CrocoPat, this is over twice as long:

```
S(x,z) := EX(y, P(x,y) & (C(y,z))
```

This can be read as follows. Compute pairs (x,z) in S by checking the existence (EX) across y of pairs (x,y) in P and pairs (y,z) in C. The EX operator can be thought of as a projection that eliminates the y column of the relation.

In my opinion, Grok's operations are higher level than CrocoPat's operators, due to the fact that Tarski's operations obviate the need for free variables. For example, the free variables in EX iterate across a range, whereas this iteration and the range are implicit in Grok. More profoundly, and perhaps controversially, I believe Tarski's notation provides better encouragement to think at a higher level and more proficient level of abstraction than does a notation with free variables.

It might make sense to add Tarski operators to CrocoPat, to take advantage of the elegance of Tarski's operators when these are sufficient, reserving use of free variables for complex or n-ary queries.

When queries become too complex, for example when searching for instances of various design patterns, Grok is no longer the appropriate language to use [10]. Beyer [3] reports that as the complexity of queries reaches the limits of the expressiveness of Grok, Grok programs become messy. Wu's [41] extensions to Grok are an attempt to solve this shortcoming.

A question that arises is: Can someone design a better Grok-like language to be close to Codd's original n-ary relational algebra, with a possibility of SQL-like notation as in GReQL, perhaps also including Tarski operators? A related question is: Why did CrocoPat invent a new n-ary notation rather than using that of GReQL?.

Deeper questions about the CrocoPat approach have to do with its concept of a *universe u*, which is a set of strings. Ideally, *u* would be the infinite set of all strings, but this is not feasible in an implemented language. As Beyer [4] says, "The finiteness and immutability of the universe are sometimes inconvenient for the developer of RML programs." This universe gives the range of CrocoPat's operators EX (exists) and FA (for all) which iterate across the members of *u*. It is also the basis of the complement (!) operator. CrocoPat chooses *u* to be the set, fixed for a given execution, of all sources and targets of inputted edges (these can only be input at program start up) as

well as all string constants in the particular CrocoPat program. If CrocoPat had string operators (which seems like a good thing) or if it could dynamically choose which files to read (which seems like a good thing), then its definition of the universe would become infeasible. Strangely, if a statement containing a string constant, such as "Hello", is added to a CrocoPat program, then the meaning of EX, FA and "!" changes in that the universe will now includes "Hello". In practice, such problems may have reasonable work-arounds, but more research seems called for to better characterize the implications of CrocoPat's universe.

There are a number of accidental (rather than inherent) differences between Grok and CrocoPat, such as the way files are handled and the means of deleting relations that may make Grok handier for manipulating big architectural graphs. For example, Grok also supports a number of features such as string handling (a weak version) and ways to search for and modify names of nodes --- which have proven to be important in architecture manipulations. It might make sense for CrocoPat to include some of these features. Grok can be used interactively, by typing in queries, or as a script languages. When used interactively, Grok is handy for browsing and exploring a database of relations, especially a database that is not well understood. By contrast, CrocoPat can only be used as a batch-oriented script language. Such differences may be resolved as these languages evolve.

8. Implementation and performance of Grok-like languages

Back in 1995-96 when I was experimenting with Tarski algebra for architectural use, it was not clear if a Grok-like language could deal with the size and complexity of problems that are interesting for architectural manipulation. This question has since been answered affirmatively, by Grok, CrocoPat and Relview. One reason for this affirmative answer is that Moore's law doubles our speed and memory about each couple of years, so our implementations get better as the years go by simply because of increasingly capable hardware.

Grok was designed to run in main memory, to optimize its speed and to simplify its implementation. The same is true of Relview, GReQL, JGrok and CrocoPat. By comparison, traditional disk-based database approaches, notably SQL, run painfully slowly when applied to large architectural graphs.

Grok uses a simple data structure to store edges. It uses three arrays, called *rel* (relation), *src* (source), and *trg* (target). Edge R(A, B) is stored in some row *i* as *rel(i) = r*, *src(i) = a*, and *trg(i) = b*, where *r, a* and *b* are 32-bit hashes of "R", "A", and "B" respectively. The simplicity of this data structure makes maintenance and enhancement of Grok particularly easy.

As Grok computes new relations, they are allocated space on the end of these growing arrays, which is compacted as relations are deleted. An operation such as relational composition, union or intersection scans the arrays to collect the edges of each operand, sorts these as appropriate for performance and then executes the actual operation as a specialized merge of the edges of the two operands. A radix sort is used, which effectively runs in time O(E) where E is the total number of edges, and the whole operation time is O(E). Transitive closure is implemented by collecting edges of the relation and shipping them to an implementation of transitive closure. This implementation of closure can easily be replaced by another better one if this is desirable. Grok is written in Turing, which is implemented by transcribing it to the C language.

JGrok's implementation is based on linked lists, in turn based on Java classes. These lists record the nodes and edges in the graph.

A dramatically different data structure is used in Relview and CrocoPat, which both use Binary Decision Diagrams (BDDs) [5]. An advantage of BDDs is that they can represent n-ary relations.

Beyer [4] states, "Experience in computer-aided verification shows that the data structure binary decision diagram (BDD) can represent even huge relations efficiently." Dong [9] is not so optimistic, saying, "research on symbolic BDD based graph algorithms seems to be worthwhile only if real applications deal with large dense graphs. The graphs which we encounter in practice, e.g., the graphs representing the network of the Internet, social networks, and railway connections, seem to be rather sparse (as they are more or less planar), however."

BDDs work best for dense graphs, as this density allows common paths in the decision diagram to be collapsed to a single path. *A priori*, it is not clear that architectural graphs are dense. In practice, the number of edges in graphs that are extracted from software commonly grows linearly with the number of nodes, which suggests sparseness rather than density.

However, Beyer gives case study measurements indicating that CrocoPat's BDD data structure takes less space than Grok's array-based data structure for large and complex queries, notably when computing the transitive closure of the CALL relation. It would be good to do more of such studies, to better determine when BDD has an advantage and to better determine what aspect of architectural graphs causes them to be efficiently encoded as BDDs. (Beyer's [3] WCRE

paper indicates that Grok crashed on memory overflows on test examples, but it appears that an obsolete version of Grok was used in those tests.)

9. Using Grok-like languages for architectural purposes

This section addresses the third question asked at the beginning of this paper: *How have Grok-like languages been used to solve problems in software architecture and software analysis?*

Grok-like languages are attractive in that they promise to let us think about and manipulate graphs in a high level, elegant and efficient way. Rather than thinking and computing on a node-by-node or edge-by-edge basis, we can deal with whole relations at a time. Alfred North Whitehead has said, "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race" [Quoted in P. Davis and R. Hersh *The Mathematical Experience*, Boston, 1981.] Also, when we embed the notation in a programming language we gain the advantage that our thinking, as written down in the notation, translates directly to an implementation.

Table 2 gives examples of architectural and software analysis problems that have been tackled using a high-level relational approach. (See also Beyer's [3] list of problems dealt with by relational means.)

As an example of the kind of work being done, van Emden [37] writes, "One of the tools we use for inferring these derived aspects is Grok, a calculator for relational algebra ... We use it, for example, to compute the 'refused bequest' smell where child classes do not use the methods that were offered by their parents." As another example, van Deursen [36] writes "One of the tools we use for inferring type relations is Grok, a calculator for relational algebra ...We use it, for example, to turn the derived type facts into the required equivalence relation."

As can be seen in this table, Feijs and his colleagues have been particularly productive in discovering and promoting ways to use a relational approach in software architecture. Feijs et al., Knodel et al. and Godfrey have been particularly active in transferring this technology to industry.

Table 2 illustrates potential benefits of using a relational approach to the analysis and manipulation of architectural structures. Ideally in the future, more and more of this kind of work will be done.

Purpose	Query	Result	Examples
1. Check compliance to architectural rules	Architectural rules	Violations of architectural rules	Holt [1996], Feijs [1998], Knodel [2008]
2. Transform graph for understandability & viewing	Lifting up along parent edges	Abstracted architectural graph	Holt [1998], Feijs [1998]
3. Find design pattern instances	Encoded design pattern	Graphs representing instances of design patterns	Consens [1989], Beyer [2003]
4. Check for conformance to a pattern	Encoded required pattern	Violations of the pattern	Guo [1999]
5. Find design anti-patterns (bad smells)	Encoded anti-pattern	Instances design of anti-pattern	van Emden [2002], Feijs [1998]
6. Find impact of change; check for recursion	Backward transitive closure of calls or uses	Impact of changes, unused functions; unexpected recursion	Feijs [1998]
7. Extract facts from source	Compute relevant facts based on parsed structure	Facts extracted from source code	Lin [2008]
8. Determine basis of (high-order) edge	Lowered versions of the edge	Edges that were lifted to create high-order edges	Fahmy [2001], Feijs [1998]
9. Find protocols (or scenario)	Encoded form of protocol for communication	Instances of communication protocols	Wu [2001]
10. Locate common, implicit types, e.g., for Y2K	Symmetric transitive closure on flow of use of types	Implicit usage of types	van Deursen [1999]

Table 2. Example uses of Grok-like languages

10. Conclusions

This paper is a retrospective based on a 1998 paper on how a high-level relational approach can formalize and automate solutions to various problems in software

architecture. Boxology and its formalization as typed graphs are an underlying part of this approach. These can help us more easily document software architecture. Grok-like languages can help us to produce understandable views of architecture. Most notably as-built architecture are amenable to this approach.

A range of future research should be pursued to explore and validate the relational approach to architecture. Languages such as Grok and CrocoPat should be evaluated and evolved. Students should be exposed to a relational approach to thinking, modelling and programming. We should work toward proposing and implementing Shaw's envisioned "robust suite of tools for analysis, consistency checking."

I hope this relational approach will increasingly help us understand software architecture, design patterns, and related concepts. Ideally this approach, with light-weight formalisms and efficient relational languages will, in the words of Northhead, guide us forward to "concentrate on more advanced problems."

Acknowledgements

Grant Weddell graciously and patiently explained database theory to me. Students in my graduate course have used Grok and helped me make it more user friendly. Mike Godfrey provided valuable suggestions to improve this paper. Thanks to Sarah Nadi for fixing various clumsiness in the paper.

11. References

[1] *ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems* http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45991

[2] Berghammer, R., Schmidt, G., "Relview --- A computer system for the manipulation of relations", in Nivat, M., Rattray, C., Rus, T., Scollo, G., editors, *Proc. 3rd Conf. on Algebraic Methodology and Software Technology (AMAST 93)*, University of Twente, The Netherlands, June 1993, Workshops in Computing, Springer, 405-406 (1993)

[3] Dirk Beyer, Andreas Noack and Claus Lewerentz, "Simple and Efficient Relational Querying of Software Structures", *Proceedings of the Tenth IEEE Working Conference on Reverse Engineering (WCRE 2003, Victoria, BC, November 13-16)*, pp. 216-225, 2003. IEEE Computer Society Press, Los Alamitos (CA).

[4] Dirk Beyer, Andreas Noack, and Claus Lewerentz, "Efficient Relational Calculation for Software," Analysis. *IEEE Transactions on Software Engineering (TSE)*, 31(2):137-149, 2005.

[5] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Trans. Computers, 35(8):677–691, 1986.

[6] Donald D. Chamberlin and Raymond F. Boyce, "SEQUEL: A Structured English Query Language," *Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control*: pp. 249–264, Association for Computing Machinery, 1974.

[7] Chikofsky, E.J. and J.H. Cross II "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software, IEEE Computer Society*: 13–17, January 1990.

[8] M. P. Consens and A. O. Mendelzon, **"**Expressing structural hypertext queries in graphlog,**"** *Proceedings of the Second Annual ACM Conference on Hypertext and Hypermedia*, Pittsburgh, Pennsylvania, United States, pp. 269 – 292, 1989.

[9] Dong, C. and Molitor P., "What graphs can be efficiently represented by BDDs?", *International Conference on Computing: Theory and Applications (ICCTA'07)*, pp. 128-134, Kolkata, India, March 2007.

[10] H. M. Fahmy, R. C. Holt and James R. Cordy, "Wins and Losses of Algebraic Transformations of Software Architectures", *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, November 26-29, 2001.

[11] L. M. G. Feijs, R. L. Krikhaar and R. C. van Ommering, "A relational approach to support software architecture analysis", *Software - Practice and Experience*, 28(4):371-400, April 1998.

[12] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong, "The software bookshelf Source", *IBM Systems Journal*, Volume 36 , Issue 4, pp. 564 – 593, 1997

[13] Brian Foote and Joseph Yoder, "Big Ball of Mud," *Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97),* Monticello, Illinois, September 1997

[14] Garlan, David and Shaw, Mary, *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, Volume 1, World Scientific Publishing Co., 1993.

[15] Ian Gorton, Essential *Software Architecture,* Springer-Verlag New York Inc., Secaucus, NJ, 2006.

[16] G. Guo, J. Atlee and R. Kazman, "A Software Reconstruction Architecture Method", *Software Architecture (Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1))*, (San Antonio, TX), February 1999, 15-33.

[17] R. C. Holt, "Binary Relational Algebra Applied to Software Architecture", CSRI Tech Report 345, University of Toronto, March 1996.

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

[19] "Introduction to the Grok Programming Language," Ric Holt, May, University of Waterloo, 2002.

[20] R. C. Holt, "Software Architecture as a Shared Mental Model", *International Workshop on Program Comprehension*, Ric Holt, Paris, June, 2002.

[21] Blind men and elephant. Jainism Global Resource Center, website Jainworld.com.

[22] Knodel, J., Muthig, D., Haury, U., Meier, G., "Architecture Compliance Checking: Experiences from Successful Technology Transfer to Industry", *12th European Conference on Software Maintenance and Reengineering* (CSMR 2008), pp. 43-52, Athens, 1-4 April 2008

[23] R. L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, Univ. of Amsterdam, The Netherlands, 1999.

[24] Kullbach, B. and A. Winter, "Querying as an Enabling Technology in Software Reengineering", *3rd European Conference on Software Maintenance and Reengineering*, IEEE Comp. Soc., 1999, 42–50.

[25] Philippe Kruchten, "Architectural Blueprints—The '4+1' View", Model of Software Architecture, *IEEE Software* 12 (6), November 1995, pp. 42-50.

[26] Lin, Yuan, *Completeness of fact extractor and a new approach to fact extraction with emphasis on Refers-to relation*, PhD thesis, University of Waterloo, 2008.

[27] Andrew J. Malton and Richard C. Holt, "Boxology of NBA and TA: A Basis for Understanding Software Architecture", Working Conference on Reverse Engineering, WCRE 2005, 8-12 Nov 2005, Pittsburgh.

[28] A.O. Mendelzon and J. Sametinger, "Reverse Engineering by Visualizing and Querying," *Software—Concepts and Tools*, vol. 16, no. 4, pp. 170-182, 1995.

[29] Tilak Mitra, "Documenting software architecture, Part 1: What software architecture is, and why it's important to document it," *IBM DeveloperWorks* 15, Apr 2008.

[30] Roger Tory Peterson, *Field Guide to the Birds,* Houghton Mifflin Company, 1934.

[31] "Definitions of 'Software Architecture'", *Software Engineering Institute*, 2008, http://www.sei.cmu.edu/architecture/published_definitions.html

[32] Mary Shaw and Paul Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Proceedings of the 21st International Computer Software and Applications Conference*, 1997, pp. 6-13.

[33] Mary Shaw and Paul Clements, "The Golden Age of Software Architecture", IEEE Software, vol 23, no 2, March/April 2006, pp. 31-39.

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

[35] Linus Torvalds, website http://www.cs.helsinki.fi/u/torvalds/

[36] A. van Deursen and L. Moonen, "Understanding COBOL systems using inferred types," *Seventh International Workshop on Program Comprehension, Proceedings*, pp. 74-81, 05/05/1999 - 05/07/1999, Pittsburgh, PA, USA, 1999.

[37] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells", *Ninth Working Conference on Reverse Engineering (WCRE 2002)*. 2002, pp. 97- 106.

[38] R.van Ommering, R.Krikhaar, and L. Feijs, "Languages for formalizing, visualizing and verifying software architectures", *Computer Languages*, Volume 27, (1/3): 3-18, 2001.

[39] Grok, Wikipedia, 2008, http://en.wikipedia.org/wiki/Grok

[40] Boxology, Wikipedia, 2008, http://en.wikipedia.org/wiki/Boxology

[41] J. Wu, A. E. Hassan and R. C. Holt. "Using graph patterns to extract scenarios," *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, pages 239–247, 2002.