Journal on Relational Methods in Computer Science, Vol. 1, 2004, pp. 73 - 92

CONSIDERING DESIGN TASKS IN OO-SOFTWARE ENGINEERING USING RELATIONS AND RELATION-BASED TOOLS

RUDOLF BERGHAMMER[1] AND ALEXANDER FRONK[2]

[1] Institute of Computer Science and Applied Mathematics
University of Kiel, D-24098 Kiel, Germany
[2] Software Technology
University of Dortmund, D-44221 Dortmund, Germany

Abstract. We demonstrate how relational algebra and its mechanization through the object-oriented Java-library KURE, which is based on the software system RELVIEW, can be used to approach practical problems in the design of object-oriented software. The examples we present range from the search of improper code pieces and 3D graphical software design through to the detection of code parts that show design pattern structure.

1 Introduction

Relations are very well-known in mathematics and computer science; the formal apparatus of relational algebra is well understood and mathematically well established. For its mechanization, there exist some tools for the computer-aided manipulation of relations and relation-algebraic formulae, as well as for relation-algebraic theorem proving. Having a look at the abundance of literature (see e.g., the references in [25, 13] or the proceedings of the past RelMiCS meetings), however, it seems that during the last decades relational algebra and the respective tools have rarely been used for real practical purposes. Apart from some exceptions, their employment was of purely academic nature. For instance, in the domain of programming and software development people investigated semantics, data structures, and algorithm development. But, as far as we know, they did not treat design patterns, maintenance of code, and software refactoring, i.e., topics which nowadays have an immense importance in practice.

Received by the editors December 03, 2003, and, in revised form, October 08, 2004.
Published on December 10, 2004.

Caused by projects like the EU COST action 274 TARSKI "Theory and Application of Relational Structures as Knowledge Instruments", presently the situation changes. People become more and more interested in problems arising from "real world" situations. This leads to a lot of work going on for practical applications of relational algebra and tools. Our article, too, goes into this direction. We assemble some practical problems occurring in object-oriented software design and discuss how relations can formally and tool-based contribute to solve them. To this end, we use an object-oriented Java-library, called KURE, which comprises the core functionality of the tool RELVIEW, a software system for calculating with relations developed at the University of Kiel since 1993. KURE has been developed as a common project of the Universities of Kiel and Dortmund. The library opens the calculus of relations to any class of tools the domain of which can be represented by relations. With this advantage, relations need no longer be created manually in the RELVIEW system; instead, a tool that shall benefit from the calculus of relations simply needs to provide a suitable mapping from its proprietary data structures to the relational data model of KURE. With these software systems at hand, relational algebra can fruitfully be integrated into object-oriented engineering-based software development. That is, both Software Engineering methods and tools are enriched by RELVIEW and/or KURE which thereby pushes further the integration of formal methods and tools that apply them.

To demonstrate the wide applicability of the relation-algebraic approach, the examples we present are taken from different areas in software design and range from the search for improper code pieces and 3D graphical software design through to the detection of code parts which indicate some design pattern structure. They are based on work done at the Universities of Kiel and Dortmund in cooperation with the Software Engineering group at the University of Cottbus and represent our most recent and prominent examples of applying relations, RELVIEW, and KURE in object-oriented software design.

This article is organized as follows. In Section 2, we provide some relation-algebraic preliminaries. Section 3 discusses different software design tasks and shows how relational algebra can support them. We briefly introduce the RELVIEW system and the KURE library in Section 4. Finally, we terminate this article with a short summary and point out some interesting further work in Section 5.

2 Relation-Algebraic Preliminaries

We assume the reader to be familiar with relational algebra. Nonetheless, we provide the notation and some definitions used throughout this article.

We write $R : X \leftrightarrow Y$ if R is a relation with domain X and range Y, i.e., a subset of $X \times Y$. If the sets X and Y of R's *type* $X \leftrightarrow Y$ are finite and of cardinality m and n, respectively, we may consider R as a Boolean matrix with m rows and n columns. Since this interpretation is well suited for many purposes and Boolean matrices are the main mean to model relations in RelView and Kure and also used to depict them graphically in RelView, we use Boolean matrix notions and notations when appropriate. Particularly, we write $R_{x,y}$ instead of $(x, y) \in R$. We write R^{T} for *transposition*, $\overline{R}$ for *complement*, $R \cup S$ for *union*, $R \cap S$ for *intersection*, $R; S$ for *composition*, and $R \subseteq S$ for *inclusion*. The *empty relation* is denoted by $\mathbb{O}$, the *universal relation* by $\mathbb{L}$, and the *identity relation* by $\mathbb{I}$.

A relation R is *univalent* if $R^{\mathsf{T}}; R \subseteq \mathbb{I}$, and *total* if $R; \mathbb{L} = \mathbb{L}$. A univalent and total relation is called a *function*. R is *injective* if R^{T} is univalent and *surjective* if R^{T} is total. A relation R is *anti-symmetric* if $R \cap R^{\mathsf{T}} \subseteq \mathbb{I}$ holds, and *transitive* if $R; R \subseteq R$ holds. The least transitive relation containing R is its *transitive closure*, R^+, which equals the union of all powers R^i where $i \geq 1$. The *reflexive-transitive closure* of R, denoted by R^*, is defined as $\mathbb{I} \cup R^+$.

The *right residual* $R \setminus S$ of two relations $R : Z \leftrightarrow X$ and $S : Z \leftrightarrow Y$ is defined as an abbreviation for the relation-algebraic term $\overline{R^{\mathsf{T}}; \overline{S}}$. From this definition it immediately follows that $R \setminus S$ has the type $X \leftrightarrow Y$, and that $(R \setminus S)_{x,y}$ holds if and only if $R_{z,x}$ implies $S_{z,y}$ for all $z \in Z$. Given two relations $R : X \leftrightarrow Y$ and $S : X' \leftrightarrow Y$, their relational sum is denoted by $R + S$. The domain of $R + S$ is the disjoint union $X + X'$, and the range is Y. If x comes from X, then $(R + S)_{x,y}$ is equivalent to $R_{x,y}$; otherwise it is equivalent to $S_{x,y}$.

To model subsets of a given set with relation-algebraic means, we use *vectors*. These are relations v with $v = v; \mathbb{L}$. For $v : X \leftrightarrow Y$, this condition means: Whatever set Z and universal relation $\mathbb{L} : Y \leftrightarrow Z$ we choose, an element $x \in X$ is either in relationship $(v; \mathbb{L})_{x,z}$ to all elements $z \in Z$ or to none. Hence, the range of a vector is irrelevant, and we may consider a vector as a relation $v : X \leftrightarrow \mathbb{1}$ with a specific singleton set $\mathbb{1} = \{\perp\}$ as its range. In this case, we omit the second subscript and write v_x instead of $v_{x,\perp}$. Such a vector can be considered as a Boolean matrix with exactly one column, i.e., as a Boolean column vector, and represents the subset $\{x \in X \mid v_x\}$ of X. A vector is said to be a *point* if it is non-empty and injective. These properties mean that it represents a singleton subset of its domain or an element from it if we identify a singleton set with the only element it contains. In the Boolean matrix model, hence, a point $v : X \leftrightarrow \mathbb{1}$ is a Boolean column vector in which exactly one component is true.

3 Using Relations in Software Engineering

Relations occur in many places in Software Engineering. For example, the pipes-and-filters architecture can be seen as a system of relations such that a filter with input X and output Y can be viewed as a relation of type $X \leftrightarrow Y$ (see, e.g., [15]). Pipes are connectors modelled by relational composition. UML class diagrams, as another example, are based on elements and relations between them, i.e., class diagrams relate classes via association, aggregation, or inheritance.

In predicate logic, it is well known that a binary predicate P over some sets X and Y can be understood as a relation R between exactly these sets: $P(x, y)$ holds if and only if the pair (x, y) is contained in R. Generally speaking, whenever aspects of a software system are modelled by means of relations, and some of its properties are captured as predicate logical formulas, these properties can also be formulated in relational algebra by transforming them into relation-algebraic terms using well-defined correspondences as, for instance, found in [25]. Theoretically, this may lead to terms involving relational descriptions of direct products in form of projection relations or so-called fork-constructions; the following list of examples, however, taken from Software Engineering, can be presented without such advanced constructs.

3.1 Detecting Improper Code Pieces

Nowadays, software systems are too large to be comprehended and maintained appropriately only by reading and re-organizing their source code. Moreover, graphical representations of the underlying code and its properties need to be provided, and, indeed, both methods and tools for the automation of many tasks in visualizing and analyzing code are developed. These tasks encompass, for example, detecting problematic code pieces, dead code, or code clones. Such analyses are vital to understanding source code, and a suitable visualization of such properties is helpful in understanding how the underlying code may be re-organized.

Structural properties of source code can be extracted from it using parsing techniques and are frequently represented by one or more relations. Then the goal is to visualize, to manipulate, and to transform these relations in order to analyze the underlying code and thereby obtain the information one is interested in. In this section, we briefly sketch how relational algebra can be used to analyze code. There also exist other specific relation-based tools for analyzing software, e.g., CrocoPat [12] (see Figure 1[1]), in which Java classes are drawn as nodes and relations as arcs between them to illustrate specific aspects of connections between Java classes, or the Grok system (see [16]).

---

[1] This picture is taken from http://www-sst.informatik.tu-cottbus.de/CrocoCosmos/gdsw.html.

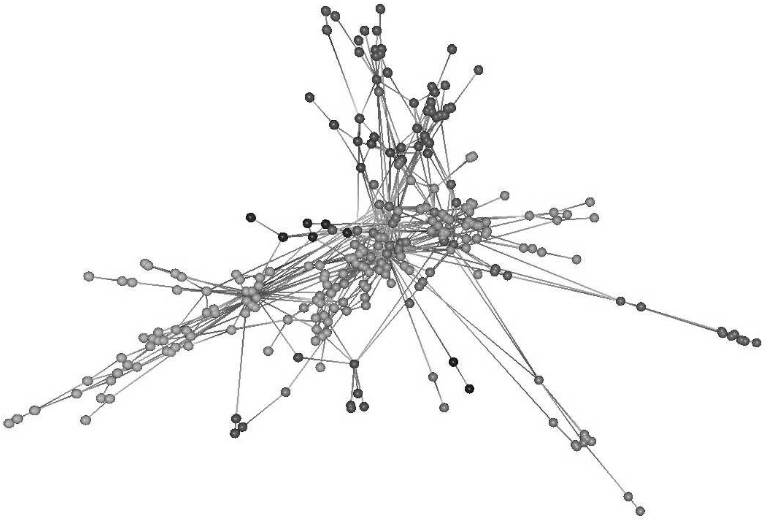

Fig. 1. Some relations between Java classes

Compared to these tools, the main benefits of RelView are, first, its very efficient implementation of relations via reduced ordered binary decision diagrams and, second, its sophisticated programming language. Binary decision diagrams allow dealing with huge relations, and due to RelView's programming language we can express many queries and manipulations in a very compact yet understandable way (see Section 4.1). With the library Kure, third, we are able to integrate the algebra of relations into any proprietary Java tool modelling with relations (cf. Section 4.2).

To discuss the detection of improper code pieces, we assume an object-oriented software system, written in the Java programming language, for instance. To keep things easy, in the following we only consider method calls and deal with analysis based on a corresponding relation calls on the set C of classes of the system:

$$\mathsf{calls} : C \leftrightarrow C \qquad\qquad \text{method calls}$$

For classes x and x', the relationship $\mathsf{calls}_{x,x'}$ holds if and only if in x there is a call of a method declared in x'.

Initial Strongly Connected Components. When analyzing an object-oriented software system, essential information can be obtained by computing the strongly connected components with respect to the relation calls. Strong connectivity means that code cannot be invoked by classes outside this component. Experience has shown that strongly connected components which additionally are initial are very often candidates for dead code.

The strongly connected components of calls are the equivalence classes of the equivalence relation comp, defined as intersection $\mathsf{calls}^* \cap (\mathsf{calls}^\mathsf{T})^*$. Using Boolean matrix terminology, the equivalence classes are described by the columns of comp such that each column assembles the (Java-)classes forming a strongly connected component. Furthermore, if the vector $c : C \leftrightarrow \mathbb{1}$ describes a single strongly connected component S, then the following calculation shows that S is initial if and only if $\mathsf{calls}; c$ is contained in c:

$$\begin{aligned}
& \forall\, x, x' : \mathsf{calls}_{x,x'} \wedge x' \in S \;\rightarrow\; x \in S \\
\Longleftrightarrow\; & \forall\, x, x' : \mathsf{calls}_{x,x'} \wedge c_{x'} \;\rightarrow\; c_x \\
\Longleftrightarrow\; & \forall\, x : (\exists\, x' : \mathsf{calls}_{x,x'} \wedge c_{x'}) \;\rightarrow\; c_x \\
\Longleftrightarrow\; & \forall\, x : (\mathsf{calls}; c)_x \;\rightarrow\; c_x \\
\Longleftrightarrow\; & \mathsf{calls}; c \subseteq c
\end{aligned}$$

Based on this fact, it is rather trivial to compute all initial strongly connected components of calls. First, one has to compute the relation comp as defined above. After that, one has to collect exactly those columns of comp that (considered as vectors c) satisfy the inclusion $\mathsf{calls}; c \subseteq c$ while avoiding duplicates. A RELVIEW implementation of this procedure is presented in Section 4.3.

Detection of Cycles. It should be possible to understand the structure of a class x without knowing which other classes use a method of x. But it is absolutely necessary to know about the methods called by methods of x — either directly or indirectly via a chain of calls of intermediate methods. This situation becomes more difficult to analyze if a chain of calls starting in x again leads back to x, i.e., if the chain forms a cycle with respect to the relation calls. Therefore, a second task frequently occurring in the analysis of software systems is to test the relation calls being cycle-free and, if this test fails, to determine the classes lying on cycles.

Since a class x lies on a cycle with respect to calls if and only if ${\mathsf{calls}^+}_{x,x}$ holds, a relation-algebraic description of cycle-freeness is rather simple:

$$\mathsf{calls} \text{ is cycle-free} \iff \mathsf{calls}^+ \subseteq \overline{\mathbb{I}}\,. \tag{1}$$

The same is true for the vector describing the set of all classes lying on a cycle. Based on (1), a little reflection yields the relation-algebraic specification

$$(\mathsf{calls}^+ \cap \mathbb{I}); \mathbb{L} : C \leftrightarrow \mathbb{1}\,. \tag{2}$$

The set described by this vector may get rather large. Experience has shown that in such a case it is advantageous to consider short cycles of a certain length, e.g., all cycles of length 10. Since x lies on a cycle of length n if and only if $\mathsf{calls}^{n}{}_{x,x}$ holds, these classes can be computed by replacing the transitive closure in the specification (2) by the n^{th} power of calls. If one wants to draw the cycles of length n as graphs, especially using RELVIEW, then this can be obtained by drawing the relation $(\mathsf{calls}^{n-1})^{\mathsf{T}} \cap \mathsf{calls}$. Using this term, for example, all such cycles contained in the code of the JWAM tool (which consists of 1133 classes and leads to a calls-relation with 3499 pairs) have been computed. Details can be found in [23].

Direct Neighborhood. To understand and visualize dependencies between the classes of a software system in general, it is reasonable to consider the direct neighborhood relation which for the calls-relation is element-wise described as follows:

$$\mathsf{calls}_{x,x'} \wedge \neg\exists\, y : \mathsf{calls}_{x,y} \wedge \mathsf{calls}^{+}{}_{y,x'} \,. \tag{3}$$

This relation is called *Hasse diagram* in the case of strict order relations. If the relation under consideration contains cycles, it has turned out to be advantageous to consider strongly connected components as hyper-vertices. In our context this means that two classes x and x' are related if and only if they are in the same strongly connected component, or if they are in the direct neighborhood relation. Relation-algebraically this is described by

$$(\mathsf{calls}^{*} \cap (\mathsf{calls}^{\mathsf{T}})^{*}) \cup (\mathsf{calls} \cap \overline{\mathsf{calls};\mathsf{calls}^{+}}) : C \leftrightarrow C \,, \tag{4}$$

where the second part of the union of (4) is an immediate consequence of the element-wise description (3) of the direct neighborhood relation.

3.2 Checking Manipulations of Design

Programming in the large is generally supported by tools allowing software design in a graphical manner. Two-dimensional representations of object-oriented classes and their relations, such as e.g. UML diagrams, serve for automatically generating code frames and hence allow manipulating code by manipulating these diagrams. In [1, 3] and [2], however, we focus on 3D representations of various relations between classes implemented in Java.

Editing 3D-diagrams requires not only to synchronize them with the underlying Java code, as is usual for 2D representations either. In addition, three-dimensional arrangements are given meaning, and thus graphical constraints need to be considered defining both the syntax and semantics of the graphical language underlying 3D representations. We declaratively define our visual language by a set of predicate logic formulas such that a 3D-diagram is a term of the language — we say: is valid — if all formulas are satisfied. Editing is seen as term replacement, i.e., editing may transfer valid diagrams into other valid diagrams. In constructive approaches to language definition using, for example, structural induction or production rules, a word or term is contained in a language if it can be constructed in a specific way. Our approach simulates this process by providing operations for editing that preserve certain constraints. This means that the validity of 3D diagrams is proved by checking the constraints imposed on the visual language. We model these constraints with relational algebra, and use RelView or Kure to efficiently check them during the editing process. In [6], we have discussed this approach in more detail.

Java Class Relations. Java software is statically structured by packages, interfaces, and classes. Packages assemble classes, interfaces, and (sub-)packages within a strict hierarchy. To keep things simple, here we only focus on packages and package containment as a relation on packages and classes.

Information cubes display package containment, and *pipes* display relations between packages (see Figure 2); the latter are of no concern here. Information cubes glue together related information and may contain arbitrary arrangements of boxes representing Java classes. Cubes are semi-transparent and allow visual access to the information within.