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 $R^\mathsf{T}$ 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 $R^\mathsf{T}$ is univalent and *surjective* if $R^\mathsf{T}$ 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 $\mathbb{1} = \{\bot\}$ as its range. In this case, we omit the second subscript and write $v_x$ instead of $v_{x,\bot}$. 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.

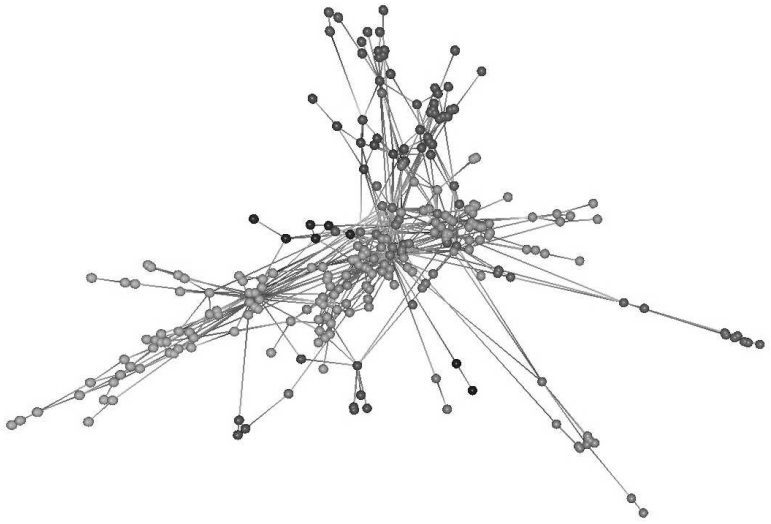

**Fig. 1.** Some relations between Java classes

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 $\mathsf{calls}$ on the set $C$ of classes of the system:

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

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 $\mathsf{calls}$. 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 $c$:

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

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 $n^{\text{th}}$ power of $\mathsf{calls}$. 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 $\mathsf{calls}$-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 $\mathsf{calls}$-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.

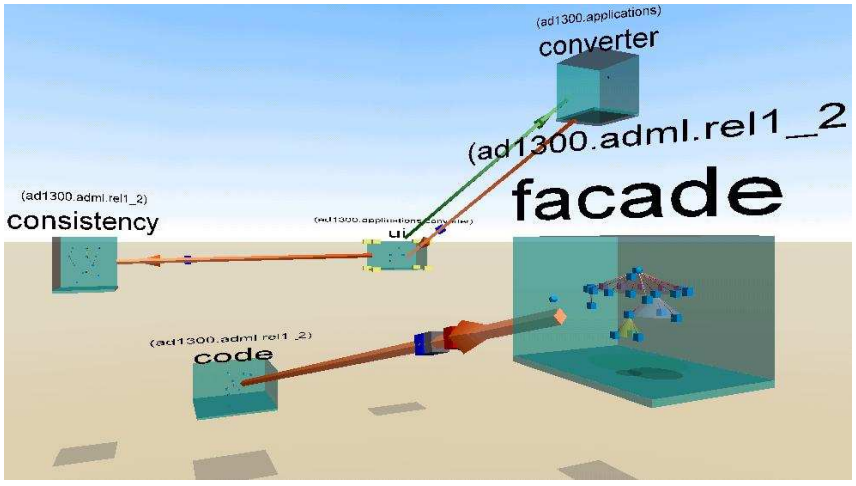


**Fig. 2.** Some related packages containing related classes

Let $C$ and $P$ be the sets of all classes and packages, respectively, of the Java code under consideration, and let $CUBE$ and $BOX$ be the disjoint sets of

information cubes and boxes, respectively. We denote the set of all graphical entities, i.e., the union of *CUBE* and *BOX*, as *ENTITY*. To relate graphical entities to Java code, we consider the following relations:

$$\begin{aligned} \mathsf{C} &: \mathit{BOX} \leftrightarrow \mathit{C} && \text{class as box} \\ \mathsf{P} &: \mathit{CUBE} \leftrightarrow \mathit{P} && \text{package as cube} \\ \mathsf{cM} &: \mathit{CUBE} \leftrightarrow \mathit{C} \cup \mathit{P} && \text{cube membership} \end{aligned}$$

The two relations $\mathsf{C}$ and $\mathsf{P}$ are univalent and total. Using the usual notation for functions, $\mathsf{C}(b)$ and $\mathsf{P}(c)$ is the Java code graphically represented by a box $b$ or a cube $c$, respectively. Since each class and package of the underlying source code is represented in the 3D diagrams, these relations are also surjective. The relation $\mathsf{cM}$ relates an information cube $c$ to a class or a package $x$ if the box or information cube representing $x$ is displayed inside the information cube.

We assume a 3D coordinate system underlying our diagrams which allows us to unambiguously determine the position of each graphical entity by means of the following relation, where *POINT* is the set of all 3D coordinates (points for short):

$$\mathsf{point} : \mathit{POINT} \leftrightarrow \mathit{ENTITY} \qquad \text{points of an entity}$$

Furthermore, we assume that different graphical entities always have different point sets and can thereby be unambiguously identified. With the help of the relation $\mathsf{point}$ we can easily define further relations to relate graphical entities displayed within a diagram. Here is a little example, saying that $e$ is inside of $e'$, which establishes the subset relation on entities:

$$\mathsf{inside}_{e,e'} :\Longleftrightarrow \forall\, p : \mathsf{point}_{p,e} \rightarrow \mathsf{point}_{p,e'} \,. \tag{5}$$

The element-wise definition (5) implies that, in relation-algebraic notation, $\mathsf{inside}$ equals the right residual $\mathsf{point} \setminus \mathsf{point}$. Using $\mathsf{inside}$, also the above-mentioned demand that different graphical entities always have different point sets can easily be described by relation-algebraic means. It is equivalent to $\mathsf{inside}$ being anti-symmetric.

**Testing Constraints.** A three-dimensional diagram needs to satisfy certain constraints describing the syntax of our 3D-diagrams and is established by translating Java code issues into graphical properties. Further, a diagram needs to be maintained during editing to determine valid 3D diagrams from which Java code can be generated. We formulate two sample constraints as predicates and transform them into relation-algebraic terms as follows.

The first constraint states that whenever a class represented by a box $b$ is a member of a package $p$ represented by a cube $c$, the box must be inside the cube or, recursively, of a cube representing a sub-package of $p$. Using the relations introduced so far and function notation for $\mathsf{C}$ and $\mathsf{P}$, this representation property reads as

$$\forall\, b, c : \mathsf{cM}_{c,\mathsf{C}(b)} \;\leftrightarrow\; \mathsf{inside}_{b,c} \vee \exists\, c' : \mathsf{cM}_{c,\mathsf{P}(c')} \wedge \mathsf{inside}_{b,c'}\,. \tag{6}$$

Applying pure relational notation, $\mathsf{cM}_{c,\mathsf{C}(b)}$ is equivalent to $\exists\, x : \mathsf{C}_{b,x} \wedge \mathsf{cM}_{c,x}$, and $\mathsf{cM}_{c,\mathsf{P}(c')}$ is equivalent to $\exists\, p : \mathsf{P}_{c',p} \wedge \mathsf{cM}_{c,p}$ due to the totality of $\mathsf{C}$ and $\mathsf{P}$. With this at hand, we transform (6) as follows:

$$\begin{aligned}
&\forall\, b, c : \mathsf{cM}_{c,\mathsf{C}(b)} \;\leftrightarrow\; \mathsf{inside}_{b,c} \vee \exists\, c' : \mathsf{cM}_{c,\mathsf{P}(c')} \wedge \mathsf{inside}_{b,c'} \\
\Longleftrightarrow\; &\forall\, b, c : (\exists\, x : \mathsf{C}_{b,x} \wedge \mathsf{cM}_{c,x}) \;\leftrightarrow\; \mathsf{inside}_{b,c} \vee (\exists\, c' : \exists p : \mathsf{P}_{c',p} \wedge \mathsf{cM}_{c,p} \wedge \mathsf{inside}_{b,c'}) \\
\Longleftrightarrow\; &\forall\, b, c : (\exists\, x : \mathsf{C}_{b,x} \wedge \mathsf{cM}^{\mathsf{T}}{}_{x,c}) \;\leftrightarrow\; \mathsf{inside}_{b,c} \vee (\exists\, c' : \exists p : \mathsf{P}_{c',p} \wedge \mathsf{cM}^{\mathsf{T}}{}_{p,c} \wedge \mathsf{inside}_{b,c'}) \\
\Longleftrightarrow\; &\forall\, b, c : (\mathsf{C}; \mathsf{cM}^{\mathsf{T}})_{b,c} \;\leftrightarrow\; (\mathsf{inside}_{b,c} \vee (\mathsf{inside}; \mathsf{P}; \mathsf{cM}^{\mathsf{T}})_{b,c}) \\
\Longleftrightarrow\; &\forall\, b, c : (\mathsf{C}; \mathsf{cM}^{\mathsf{T}})_{b,c} \;\leftrightarrow\; (\mathsf{inside} \cup (\mathsf{inside}; \mathsf{P}; \mathsf{cM}^{\mathsf{T}}))_{b,c} \\
\Longleftrightarrow\; &\mathsf{C}; \mathsf{cM}^{\mathsf{T}} = \mathsf{inside} \cup \mathsf{inside}; \mathsf{P}; \mathsf{cM}^{\mathsf{T}}
\end{aligned}$$

That is, the representation property (6) holds if and only if the above equation holds, which can simply and efficiently be tested by RELVIEW or KURE (see Section 4.3).

The second constraint demands that each box $b$ be contained in at most one cube $c$, i.e., each Java class is a member of at most one package. An obvious formalization of this containment property using the cardinality operator on sets is

$$\forall\, b : |\{c : \mathsf{cM}_{c,\mathsf{C}(b)}\}| \leq 1\,. \tag{7}$$

To transform property (7) into a relation-algebraic version, we again replace the relationship $\mathsf{cM}_{c,\mathsf{C}(b)}$ by $\exists\, x : \mathsf{C}_{b,x} \wedge \mathsf{cM}_{c,x}$ and then proceed as follows:

$$\begin{aligned}
&\forall\, b : |\{c : \mathsf{cM}_{c,\mathsf{C}(b)}\}| \leq 1 \\
\Longleftrightarrow\; &\forall\, b : |\{c : \exists\, x : \mathsf{C}_{b,x} \wedge \mathsf{cM}_{c,x}\}| \leq 1 \\
\Longleftrightarrow\; &\forall\, b : |\{c : \exists\, x : \mathsf{C}_{b,x} \wedge \mathsf{cM}^{\mathsf{T}}{}_{x,c}\}| \leq 1 \\
\Longleftrightarrow\; &\forall\, b : |\{c : (\mathsf{C}; \mathsf{cM}^{\mathsf{T}})_{b,c}\}| \leq 1 \\
\Longleftrightarrow\; &(\mathsf{C}; \mathsf{cM}^{\mathsf{T}})^{\mathsf{T}}; (\mathsf{C}; \mathsf{cM}^{\mathsf{T}}) \subseteq \mathbb{I}
\end{aligned}$$

That is, the containment property (7) holds if and only if $\mathsf{C}; \mathsf{cM}^{\mathsf{T}}$ is a univalent relation.

### 3.3 Searching for Design Patterns

The construction of reusable software requires to provide a well-structured and maintainable design. For this purpose, Gamma et al. propose design patterns (see [20] for details). A pattern factors out structural aspects of the software under construction and arranges the classes involved in the pattern in a specific way. Patterns are based on experience and hence describe reoccurring problems.

It is also clear that designing software is a non-trivial matter, and design needs to be reconsidered during software development more than once. Reengineering, i.e., changing a design into a more maintainable one (as explained e.g., in [26]), or refactoring, i.e., enhancing a design without altering the external behavior of the code (cf. [17]), are such tasks a software engineer may face during the development process. The re-organization of software may thus comprise the replacement of parts of a design by an arrangement of classes following a design pattern. Since the classes under replacement may be scattered, both a suitable layout of a design document and experience in code inspection are vital for locating these classes.

Again, methods and tools to automatically support this search may help in this situation. The automation requires a formal description of both the design patterns and the software design documents given by a set of UML class diagrams, for instance. It is then possible to analyze the latter and search for classes indicating possible occurrences of the former.

Considering structural properties as was done in the previous applications, a design pattern can be formalized by a set of relations in analogy to software systems, since their structure is usually depicted by a UML class diagram as well. As a consequence, finding structures that match a design pattern can also be carried out by relation-algebraic means and our systems.

**Formalizing Design Patterns.** We model a design pattern by means of relations and express its structural properties in relation-algebraic terms. Such properties are necessary but not sufficient to identify classes forming a pattern, or to detect classes indicating that they could be re-organized as a pattern. Moreover, code needs to be inspected since the methods of participating classes need to be considered as well to decide if a situation matches a pattern. Nonetheless, analyzing relations between classes in this way helps to detect suspicious parts of the design worth examining in more detail.

Our approach thus works in three steps, the first two of which are treated by relational algebra. The first step requires to formalize the structure of a pattern by means of relations. We call this step *pattern formalization*. The second step filters out (sub-)relations representing the formalization of a software design which have exactly the properties a design pattern requires. This step is called

*pattern indication*. The third step, called *pattern ascertainment*, is not automated in this way since it requires to pairwise compare the methods defined in these classes to ascertain that the desired pattern was indeed found. In comparison to approaches for instance found in [21], we both formalize and analyze design patterns by means of relations only.

We study the Composite pattern as an example to explain the first two steps of our approach.

**The Composite Pattern.** Figure 3 shows the Composite pattern. This pattern consists in essence of a class *Component* and its two subclasses, *Composite* and *Leaf*. A composite aggregates components as its children which, recursively, may be composites or leafs. Graph-theoretically, the objects involved in a composition form a tree with respect to inheritance.

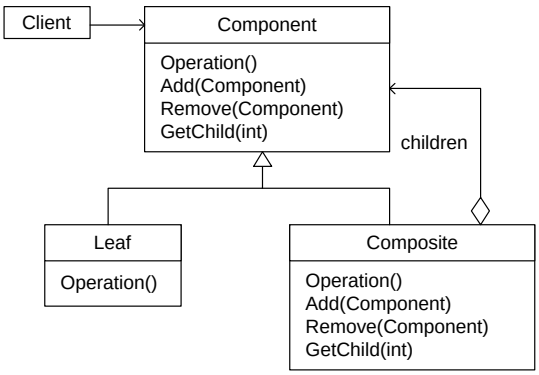


**Fig. 3.** The Composite Pattern

The structurally interesting part of this pattern is given by the inheritance relation and aggregation between the classes *Component* and *Composite*. Hence, we need to consider the following two relations:

$$\begin{aligned} \mathsf{inherits} &: C \leftrightarrow C && \text{class inheritance} \\ \mathsf{aggregates} &: C \leftrightarrow C && \text{class aggregation} \end{aligned}$$

For classes $x$ and $x'$ the relationship $\mathsf{inherits}_{x,x'}$ holds if and only if $x$ inherits from $x'$, and the relationship $\mathsf{aggregates}_{x,x'}$ holds if and only if $x$ aggregates $x'$.

The structure of the Composite pattern leads to the task to test for two classes $a$ and $b$ whether $a$ aggregates $b$, class $a$ is a subclass of $b$, and whether there exists a further subclass $c$ of $b$ at the same time. If this is true, then $a$ is a candidate for the composite, $b$ for the corresponding component, and $c$ for the

leaf. An element-wise formalization of the properties $a$ and $b$ have to fulfill is:

$$\mathsf{aggregates}_{a,b} \wedge \mathsf{inherits}_{a,b} \wedge \exists\, c : a \neq c \wedge \mathsf{inherits}_{c,b}\,. \tag{8}$$

Using the definition of relational composition and the identity relation, formula (8) evolves to the following relation:

$$\mathsf{aggregates} \cap \mathsf{inherits} \cap \overline{\mathbb{I}}; \mathsf{inherits} : C \leftrightarrow C\,. \tag{9}$$

This relation enumerates as its members exactly all pairs $a$ and $b$ which are candidates for the COMPOSITE pattern.

Notice, however, that this relation represents "direct matches" of the pattern. But the situation is not always that simple. The second step, *pattern indication*, is used to weaken the above test and to search for pairs of classes that do not directly match condition (8). For example, there might exist a class $x$ that inherits from a class $a$ inheriting from a class $b$ which in turn is aggregated by $x$. In this case, $b$ and $x$ form a composition via the class $a$. Variations of this theme can be found using the inheritance relation, and, of course, any arbitrary mixture of these two situations. The pattern indication step hence formulates these possibilities relation-algebraically by simply using the transitive closures $\mathsf{aggregates}^+$ and $\mathsf{inherits}^+$ instead of $\mathsf{aggregates}$ and $\mathsf{inherits}$ in (9). Finding candidates that possibly match a pattern is done in these two steps, and the members of relation (9) and its variations can be further analyzed in the third step, the *pattern ascertainment*.

## 4 Implementation with RelView and Kure

In this section, we introduce the well-known system RELVIEW and the library KURE based on this system. Since KURE is an extraction of the functional core of RELVIEW (it provides the functionality of the latter tool except for the graphical user interface), almost all properties known for RELVIEW carry over to KURE. Especially, KURE uses the same implementation of relations via reduced ordered binary decision diagrams and offers the same programming language and file-handling possibilities as RELVIEW does. In addition to RELVIEW, KURE is delivered as an object-oriented Java library and thus allows for its employment in any proprietary Java tool.

### 4.1 The RelView System

RELVIEW (see, e.g., [4, 5, 8, 10]) is a software system for calculating with relations and relation-algebraic programming. In it, all data are represented as relations,

which the system visualizes in different ways. It offers several algorithms for pretty-printing a relation for which domain and range coincide as a directed graph. Alternatively, an arbitrary relation may be displayed as a Boolean matrix, which is very useful for visual editing and also for discovering structural properties that are not evident from a graphical presentation. Because RelView often works on large data, it uses a very efficient implementation of relations based on reduced ordered binary decision diagrams. For a more detailed treatment of this topic, see [23].

RelView can manage as many relations simultaneously as memory allows and the user may manipulate and analyze them by pre-defined operations and tests, relational functions, and relational programs. The pre-defined operations include all operations presented so far, where the symbols for the basic operations of relational algebra are `^` (transposition), `-` (negation), `|` (union), `&` (intersection), and `*` (composition). The pre-defined tests include e.g., `incl`, `eq`, and `empty` for testing inclusion, equality, and emptiness of relations, respectively.

All these operations and tests can be accessed through simple mouse-clicks, but the usual way is to compose them within relational functions and programs. A declaration of a relational function is of the form $F(X_1, \ldots, X_n) = t$, where $F$ is the function name, the $X_i$, $1 \leq i \leq n$, are the formal parameters (standing for relations), and $t$ is a relation-algebraic term over these parameters and the relations currently in the system's workspace. A relational program in RelView essentially is a while-program based on the datatype of relations. Such a program has many similarities with a function procedure in Pascal or Modula-2. It starts with a headline containing the program's name and a list of formal parameters. Then the declaration part follows, which consists of the declarations of local relational domain constructions (direct products and sums), local relational functions, and local relational variables. The third part of a relational program is its body, a sequence of statements which are separated by semicolons and terminated by the return-clause.

### 4.2 The Kure Library

We recently implemented the algebra of relations as a data type provided in an object-oriented Java-library, called Kure, which extracts the core functionality of the tool RelView. Based on a former C version (see [23]) we developed the Java version of Kure mainly in the course of the diploma thesis [27] at the University of Dortmund, but, of course, in close cooperation with the Kiel RelView-group. In Kure, relations are treated as objects: Elements in relations are addressed by *get-* and *set*-methods, terms are evaluated using a *evaluateTerm*-method. Relations are handled by a so called *relation manager*. Therein, functions can

be defined using a *createFunction*-method, programs and pre-defined functions are imported using a *loadProgram*-method. Additionally, Kure uses *exception*-handling.

The library allows us to easily employ the algebra of relations in any Java-based tool modelling with relations. With this benefit, relations need no longer be created manually within the RelView system; nor need they be encoded as an adjacency matrix, list of pairs, or some other standard representation by a proprietary tool, then imported into RelView via the system's ASCII file format in order to calculate with these relations, and results eventually reimported into one's own tool for further use. Moreover, a developer simply needs to deduce suitable relations from the data structure used in his tool and can then seamlessly use the algebra of relations therein. Kure thus allows a developer to examine even large relations modelling, for instance, some source code being analyzed, since these relations can be generated automatically by the respective tool and need not be created manually within the RelView system or by an auxiliary tool outside of it.

As a working example, we employed Kure in a Petri net CASE tool, called PetRA (see [19]). As shown in [18], the PetRA system can both model Petri nets interactively on the screen and analyze them relation-algebraically. Results of running a relational program are drawn into the Petri net displayed on the screen. This offers a nice integration of Petri net modelling with relational analysis in a graphical manner. A main task of such a tool concerning relations is to transform the graphical net representation into a set of relations handleable by Kure. Results of running a relational program are re-transformed into the graphical net model. We thus strongly integrated the data model of PetRA with the data model *relation* by a model transformation process. Due to Kure, the relational aspects of Petri net analysis are hidden from the user which opens PetRA for any user interested in Petri net analysis.

### 4.3 Relational Algebra in RelView and Kure

Because of the expressiveness of the programming language and the very efficient implementation of relations using reduced ordered binary decision diagrams, the RelView system (respectively the Kure library) is an excellent tool for executing many relational descriptions, not only when performing software design tasks. In the following, we will formulate three of the relational descriptions we have developed in Section 3 as RelView code.

As a first example, we consider the relation-algebraic version of the representation property (6). In the syntax of RelView, it looks as follows:

```
eq(C * cM^, inside | inside * P * cM^) .
```

Hence, deciding property (6) means to evaluate this RELVIEW term and to look whether its value is *true*, which in RELVIEW is represented by a universal relation on a singleton set, or the value is *false*, which is represented by the corresponding empty relation.

Next, we consider the relation-algebraic description (2) of the set of all classes that lie on a cycle with respect to the relation calls. The programming language of RELVIEW contains a pre-defined operation `trans` for computing the transitive closure $R^+$ of a relation $R$ and a pre-defined operation `dom` for computing the vector $R;\mathbb{L}$ describing the set of elements from $R$'s domain which are related with an element of $R$'s range. Also many constant relations can be computed via calls of pre-defined RELVIEW operations. Especially, the call `I(R)` computes the identity relation of type $X \leftrightarrow X$ for a relation $R : X \leftrightarrow X$. An abstraction from (2) to arbitrary relations, hence, yields the following RELVIEW function for computing the set of all elements of the carrier set of $R$ lying on a cycle with respect to $R$:

```
oncycle(R) = dom(trans(R) & I(R)) .
```

Our third example deals with the computation of the initial strongly connected components of the relation calls. Here we start with the following RELVIEW program, where the pre-defined operations `Ln1` and `On1` compute for the relation $R : X \leftrightarrow X$ the universal vector $\mathbb{L} : X \leftrightarrow \mathbb{1}$ and the empty vector $\mathbb{O} : X \leftrightarrow \mathbb{1}$, respectively, the pre-defined operation `point` yields one of the points contained in a non-empty vector, and the pre-defined operation `+` computes the relational sum:

```
classes(R)
  DECL C, v, c
  BEG  C = On1(R);
       v = Ln1(R);
       WHILE -empty(v) DO
         c = R * point(v);
         IF isempty(C) THEN
           C = c
         ELSE
           C = (C^ + c^)^ FI;
         v = v & -c OD
       RETURN C
  END
```

Using Boolean matrix terminology, the columns of the result `C` of the program `classes` are pair-wise disjoint, each column of `C` describes an equivalence class of the input `R`, and each equivalence class of the input is described by a column of `C`. The program `classes` is very similar to the relational program for computing equivalence classes which is formally developed in [8] based on relation-algebraic

descriptions of the three just-mentioned properties. Its correctness can be shown by an adaptation of the derivation found in [8].

It is an easy exercise to refine `classes` to a RELVIEW program, `initscc` say, such that the refined program computes the initial strongly connected components of the relation calls. The refinement process consists of two parts.

Since the strongly connected components of calls are the equivalence classes of the equivalence relation $\mathsf{calls}^* \cap (\mathsf{calls}^\mathsf{T})^*$, in a first step we rename the formal parameter `R` into `calls`, declare the former parameter `R` now as an additional local relational variable, declare a new local relational function

```
rtc(S) = I(S) | trans(S)
```

for computing reflexive-transitive closures, and insert the additional assignment `R = rtc(calls) & rtc(calls^)` in front of the hitherto first assignment `C = On1(R)` of the program's body. After these changes, the program computes all strongly connected components of the input `calls` as columns of its result `C`.

In a second step, now we alter the program obtained so far in such a way that during the run through the while-loop only the initial strongly connected components of the input `calls` are collected as columns of the result `C`. This is obtained by simply checking after the computation of the next equivalence class `c` via the assignment `c = R*point(v)` whether `c` is initial and executing only in that case the conditional of the original while-loop. Since the RELVIEW formulation `incl(calls*c,c)` of the initiality test is an immediate consequence of what was said in Section 3.1, we arrive at the RELVIEW programm

```
initscc(calls)
  DECL rtc(S) = I(S) | trans(S);
       R, C, v, c
  BEG  R = rtc(calls) & rtc(calls^);
       C = On1(R);
       v = Ln1(R);
       WHILE -empty(v) DO
         c = R * point(v);
         IF incl(calls*c,c) THEN
           IF isempty(C) THEN
             C = c
           ELSE
             C = (C^ + c^)^ FI FI;
         v = v & -c OD
       RETURN C
  END
```

for a column-wise computation of the initial strongly connected components.

## 5 Summary

In this article we have demonstrated how relations encompass important aspects of software design and thereby have integrated relational algebra with an important Software Engineering task. We have presented some examples studied at Kiel University and Dortmund University in cooperation with Cottbus University: code inspection supported by relational analysis, checking validity of three-dimensional diagrams for representing and manipulating software structures, and, last but not least, an approach to detecting parts of a software design indicating some design pattern structure. In all these cases, the tool-supported application of a formal method by combining relational algebra with RelView and/or Kure has turned out to be of practical benefit in software design and allows one to argue formally about software issues.

The RelView system operationalizes relational algebra and allows one to visualize and analyze even large relations in short time due to its very efficient implementation of relations using binary decision diagrams. Beyond software design, relational algebra supports other tasks of computer science equally well. For instance, graphs, their traversal to solve reachability and related problems, and many properties such as cycle-freeness or connectedness can advantageously be modelled, specified, algorithmically solved etc. by relations (see, e.g., [25, 13, 24, 8, 9, 22]). Petri nets and their structural and dynamic properties also can be analyzed by means of relational algebra; here [10, 18] present some applications. As last applications we mention specifying with relations for prototyping, as described in [7], and the definition of semantics of programming languages, as done in [11, 14]. Summing up: We believe that relational algebra offers powerful both descriptive and analytical mechanisms. With Kure, in many cases handling relations becomes more flexible than with RelView and relation algebra is opened for any tool modelling with relations.

**Acknowledgements.** We thank Ulf Milanese for many valuable discussions on the topic of the article and the referees for their helpful comments and suggestions.

## References

1. K. Alfert and A. Fronk. 3-Dimensional Visualization of Java Class Relations. In *Proceedings of the 5th World Conference on Integrated Design Process Technology*. Society for Design and Process Science, 2000. Published on CD-ROM.
2. K. Alfert and A. Fronk. Manipulation of 3-dimensional Visualization of Java Class Relations. In *Proceedings of the 6th World Conference on Integrated Design Process Technology*. Society for Design and Process Science, 2002.

3. K. Alfert, A. Fronk, and F. Engelen. Experiences in 3-dimensional Visualization of Java Class Relations. *Transactions of the SDPS: Journal of Integrated Design and Process Science*, 5(3):91–106, September 2001.
4. R. Behnke, R. Berghammer, T. Hoffmann, B. Leoniuk, and P. Schneider. Applications of the RelView System. In *Tool support for system specification, development and verification*, Advances in Computer Science, pages 33–47. Springer, 1998.
5. R. Behnke, R. Berghammer, E. Meyer, and P. Schneider. RelView — A System for Calculating with Relations and Relational Programming. In *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science (LNCS)*, pages 318–321. Springer, 1998.
6. R. Berghammer and A. Fronk. Applying Relational Algebra in 3D Graphical Software Design. In *Relational and Kleene-Algebraic Methods in Computer Science*, volume 3051 of *Lecture Notes in Computer Science (LNCS)*, pages 62–73. Springer, 2003.
7. R. Berghammer, T. Hoffmann, B. Leoniuk, and U. Milanese. Prototyping and Programming with Relations. *Electronic Notes in Theoretical Computer Science*, 44(3):24 pages, 2003.
8. R. Berghammer and T. Hoffmann. Modeling Sequences within the RelView System. *Journal of Universal Computer Science*, 7(2):107–123, 2001.
9. R. Berghammer and T. Hoffmann. Relational Depth-first-search with Applications. *Information Sciences*, 139(3-4):167–186, 2001.
10. R. Berghammer, B. von Karger, and C. Ulke. Relational-algebraic Analysis of Petri Nets with RelView. In *Proceedings of the 2nd Workshop on Tools and Applications for the Construction and Analysis of Systems (TACAS '96)*, number 1055 in LNCS, pages 49–69. Springer, 1996.
11. R. Berghammer and H. Zierer. Relational Algebraic Semantics of Deterministic and Nondeterministic Programs. *Theoretical Computer Science*, 43:123–147, 1986.
12. D. Beyer and C. Lewerentz. CrocoPat: Efficient Pattern Analysis in Object-oriented Programs. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003)*, pages 294–295. IEEE Computer Society Press, May 2003.
13. C. Brink, W. Kahl, and G. Schmidt, editors. *Relational Methods in Computer Science*. Advances in Computing Science. Springer, 1997.
14. J. Desharnais amd A. Mili and T. T. Nguyen. *Relational Methods in Computer Science*, chapter Refinement and Demonic Semantics, pages 166–183. In Brink et al. [13], 1997.
15. E.-E. Doberkat. Pipelines: Modelling a Software Architecture through Relations. *Acta Informatica*, 40:37–79, 2003.
16. H. M. Fahmy, R. C. Holt, and J. R. Cordy. Wins and Losses of Algebraic Transformations of Software Architecture. In *IEEE 16th International Conference on Automated Software Engineering*, November 2001.
17. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
18. A. Fronk. *Using Relation Algebra for the Analysis of Petri Nets in a CASE Tool Based Approach*. In *Proceeedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM), Beijing*. pages 396–405, IEEE Computer Society Press, September 2004.
19. A. Fronk and J. Pleumann. Relationenalgebraische Analyse von Petri-Netzen: Konzepte und Implementierung, In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN)*, pages 61–68. Technical report tr-ri-05-251, Universität Paderborn, Germany, September 2004.
20. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.
21. D. Heuzeroth, Th. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *Proceedings of the 11th International Workshop on Program Comprehension, Portland*. IEEE, May 2003.
22. T. Hoffmann. *Fallstudien relationaler Programmentwicklung am Beispiel ausgewählter Graphdurchlaufstrategien*. PhD thesis, Universität Kiel, Logos Verlag, 2002.
23. U. Milanese. *Zur Implementierung eines ROBDD-basierten Systems für die Manipulation und Visualisierung von Relationen*. PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, 2003.

24. J. Ravelo. Two Graph-algorithms derived. *Acta Informatica*, 26:489–510, 1999.
25. G. Schmidt and T. Ströhlein. *Relations and Graphs.* EATCS Monographs on Theoretical Computer Science. Spinger, 1993.
26. Ian Sommerville. *Software Engineering.* Addison-Wesley, 2000. 6th Edition.
27. O. Szymanski. Relationale Algebra im dreidimensionalen Software-Entwurf — ein werkzeugbasierter Ansatz. Master's thesis, Universität Dortmund, 2003.

Journal on Relational Methods in Computer Science, Vol. 1, 2004, pp. 73 - 92
Received by the editors December 03, 2003, and, in revised form, October 08, 2004.
Published on December 10, 2004.

This article may be accessed via WWW at `http://www.jormics.org`.