Machine Support of Relational Computations: The Kiel RELVIEW* System

Ralf Behnke, Rudolf Berghammer, and Peter Schneider

{rb,rub,psc}@informatik.uni-kiel.de

Bericht Nr. 9711

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preusserstraße 1–9
D–24105 Kiel
Germany

Abstract

People working with relations and graphs very often use a greater or smaller example and manipulate it with pencil and paper in order to prove or disprove some property or to obtain an impression how a certain algorithm works. For supporting such a task by machine, the RELVIEW system has been constructed. This report is intended as a user's and programmer's guide for RELVIEW. But it informs also about relational algebra, the theoretical background behind the system.

*WWW: http://www.informatik.uni-kiel.de/~progsys/relview.html

Contents

1 Introduction

The calculus of binary relations has its roots in the second half of the 19th century with the pioneering work of A. de Morgan, C.S. Peirce, and E. Schröder. The modern algebraic development of binary relations starts with to A. Tarski and his co-workers, see [22, 8, 16]. In the last two decades relational algebra has been accepted by many mathematicians and computer scientists as a convenient formalism for describing fundamental concepts of, e.g., graphs, combinatorics, lattices, games, and Computer Science (like relational semantics, program correctness, and data bases).

Relational algebra has a fixed and surprisingly small set of operations. On finite carriers, all operations easily can be implemented, and, thus, a computer system supporting relational computations easily can be implemented, too. Such computations may be used for constructing examples and counter examples to prove resp. disprove a property, or for obtaining an impression how a certain relation-algorithmic algorithm works. In this paper we describe such a relation-based computer system for visualization, analysis, and manipulation of discrete structures, called RELVIEW. Written in the C programming language, it runs under X windows and makes full use of the graphical user interface. Currently the system is used in about 30 installations all over the world.

In RELVIEW all data are represented as binary relations, which the system visualizes in two different ways. For homogeneous relations, RELVIEW offers a representation as directed graphs, including several different algorithms for pretty-printing. As an alternative, an arbitrary relation may be displayed on the screen as a Boolean matrix. This is often very useful for visual editing and also for discovering various structural properties that are not evident from the graph representation. The RELVIEW system can manage as many graphs and matrices simultaneously as memory allows and the user may manipulate and analyse the relations behind these objects by combining them with the operators of relational algebra. The elementary operations can be accessed through simple mouse-click, but they can also be combined into relational expressions, mappings, and imperative programs. Relations and graphs can be stored. Functions and programs can be stored as well and applied to many sets of input data. Frequently, RELVIEW is used for prototyping. Then the system often works on large objects, e.g., if a membership relation or another higher-order object appears during a computation (see [5, 6]). For that reason, it uses a very efficient and sophisticated internal representation of relations as well as very efficient implementations of the relational basic operations.

The first versions of RELVIEW have been written at the University of the German Forces Munich from 1988 until 1992; see [1, 3, 4]. Based on the experiences with the Munich system, in the last three years RELVIEW was redesigned and extended at Kiel University, and now Kiel University is responsible for the further development. This document gives a description of the present RELVIEW version 6.2, inclusive a user's manual and some implementation details, and informs also about the theoretical background. Concretely, it is organized as follows. In Section 2, we present the basic concepts of relational algebra which are necessary for advanced working with RELVIEW. Section 3 deals with the system. Firstly, we give an overview. Then we describe the system's graphical user interface, i.e., the windows and command buttons, in detail. Next, we focus our attention to computations using relational expressions and programs. After that, we describe the labeling mechanism of RELVIEW. And, finally, we deal with some miscellaneous topics like the configuration of the system, the use of a start-up file, and the installation of RELVIEW on a host system. Section 4 demonstrates the use of RELVIEW by means of some examples. We consider a lattice-theoretic application, solve some problems on Petri nets, and deal also with graph-theoretic algorithms. We conclude the document with some remarks on future work on RELVIEW.

2 Relation-Algebraic Preliminaries

RELVIEW is a computer system for the manipulation of relations, more general, for relational programming. Therefore, it is necessary to know about the basic concepts of this field to be able to work with. In this section we give a short introduction to relational algebra. For more details concerning the algebraic theory of relations, see e.g., [8, 16, 21].

2.1 Axiomatic Relational Algebra

A *typed relation* $R : X \leftrightarrow Y$ consists of a domain X, a range Y and a set $R \subseteq X \times Y$. X and Y are also called the carrier sets of R. The set of all (typed) relations with domain X and range Y is denoted by $[X \leftrightarrow Y]$. When the type of a relation is clear, we abbreviate $R : X \leftrightarrow Y$ to R. If the sets X and Y are finite and of cardinality m and n, respectively, then we may consider R as a Boolean matrix with m rows and n columns. Since this Boolean matrix interpretation is well suited for a graphical representation, and also used in RELVIEW, we use matrix notation and write R_{xy} instead of $(x, y) \in R$.

We assume the reader to be familiar with the basic operations on relations, viz. R^{T} (transposition, conversion), $\overline{R}$ (negation, complement), $R \cup S$ (join, union), $R \cap S$ (meet, intersection), $R \cdot S$ (composition, multiplication; often abbreviated by $R\,S$), $R \subseteq S$ (inclusion), and the special relations O (empty relation), L (universal relation), and I (identity relation). The set-theoretic operations $\overline{}$, $\cup$, $\cap$, the ordering $\subseteq$, and the constants O and L form a Boolean lattice. Some further well-known rules concerning relations are, for instance,

$$
\begin{array}{ll}
R^{\mathsf{T}\mathsf{T}} = R & R \subseteq S \Longrightarrow R^{\mathsf{T}} \subseteq S^{\mathsf{T}} \\
(R\,S)^{\mathsf{T}} = S^{\mathsf{T}}\,R^{\mathsf{T}} & \overline{R^{\mathsf{T}}} = \overline{R}^{\mathsf{T}} \\
R \subseteq S \Longrightarrow Q\,R \subseteq Q\,S & R \subseteq S \Longrightarrow R\,Q \subseteq S\,Q \\
Q\,(R \cap S) \subseteq Q\,R \cap Q\,S & Q\,(R \cup S) = Q\,R \cup Q\,S \\
(R \cap S)^{\mathsf{T}} = R^{\mathsf{T}} \cap S^{\mathsf{T}} & (R \cup S)^{\mathsf{T}} = R^{\mathsf{T}} \cup S^{\mathsf{T}}\,,
\end{array}
$$

where the last two lines also hold if binary meet and join are replaced by arbitrary meet (greatest lower bound, denoted by $\bigcap_i R_i$) and join (least upper bound, denoted by $\bigcup_i R_i$). The theoretical framework for all these rules to hold is that of an (axiomatic) relational algebra. As constants and operations of this abstract algebraic structure we have those of concrete (i.e., set-theoretic) relations. The axioms of relational algebra are

1. the axioms of a complete Boolean lattice for negation, join, meet, the ordering, and the empty and universal relation,

2. the axioms of a monoid for composition and the identity relation,

3. the so-called *Dedekind rule*

$$Q\,R \cap S \subseteq (Q \cap S\,R^{\mathsf{T}})\,(R \cap Q^{\mathsf{T}}\,S)\,,$$

4. and the so-called *Tarski rule*

$$R \neq \mathsf{O} \iff \mathsf{L}\,R\,\mathsf{L} = \mathsf{L}\,.$$

Usually, in the latter rule only the "$\Longrightarrow$" direction is demanded. As an immediate consequence of our version of the Tarski rule, we avoid the degenerate case $\mathsf{O} = \mathsf{L}$ like [22] does. The inequation $\mathsf{O} \neq \mathsf{L}$ implies that in our approach domain X and range Y of a concrete relation $R : X \leftrightarrow Y$ are non-empty. This is very helpful for defining properties on relations in a component-free manner (see below) and also agrees exactly with the practical use of relations.

From the Dedekind rule we obtain the so-called *Schröder equivalences* (also known as "Theorem K" of A. de Morgan), viz.

$$Q\,R \subseteq S \iff Q^{\mathsf{T}}\,\overline{S} \subseteq \overline{R} \iff \overline{S}\,R^{\mathsf{T}} \subseteq \overline{Q}$$

which are in fact equivalent with the Dedekind rule.

2.2 Some Specific Classes of Relations

The basic operations and constants mentioned in Section 2.1 are very helpful for defining simple properties on relations. In the following, we consider some well-known classes of relations and define them in a component-free manner. Corresponding tests are also implemented in the RELVIEW system.

2.2.1 Orderings and Equivalences

A relation $R : X \leftrightarrow X$, i.e., a relation for which domain and range coincide, is called *homogeneous*. Without reference to domain and range we have that R is homogeneous if and only if the product $R\,R$ is defined. In the Boolean matrix model of relations, a homogeneous relation is quadratic.

Two important classes of homogeneous relations are the following: A relation R is said to be *reflexive* if $\mathsf{I} \subseteq R$, *transitive* if $R\,R \subseteq R$, and *antisymmetric* if $R \cap R^{\mathsf{T}} \subseteq \mathsf{I}$. By a *partial ordering* we mean a reflexive, antisymmetric, and transitive relation. Another important class of homogeneous relations are *equivalence relations* which are reflexive, transitive and *symmetric*, where the latter property holds for R if $R \subseteq R^{\mathsf{T}}$.

2.2.2 Mappings, Homomorphisms, and Isomorphisms

An arbitrary (also called *heterogeneous*) relation $R : X \leftrightarrow Y$ is said to be a *partial mapping* or, briefly, to be *univalent* if $R^{\mathsf{T}}\,R \subseteq \mathsf{I}$, and R is said to be *total* if $R\,\mathsf{L} = \mathsf{L}$, which is, in turn, equivalent to $\mathsf{I} \subseteq R\,R^{\mathsf{T}}$. For a univalent relation Q we have the distributivity law $Q\,(R \cap S) = Q\,R \cap Q\,S$, where we are also allowed to replace binary meet by arbitrary meet. As usual, a univalent and total relation is said to be a (total) *mapping*. A relation R is called *injective* if R^{T} is univalent and *surjective* if R^{T} is total. An injective and surjective relation is said to be *bijective*.

Let $R : X_1 \leftrightarrow Y_1$ and $S : X_2 \leftrightarrow Y_2$ be two relations and consider a pair $\mathcal{H} = (\Phi, \Psi)$ of mappings $\Phi : X_1 \leftrightarrow X_2$ and $\Psi : Y_1 \leftrightarrow Y_2$. The pair $\mathcal{H}$ is called a *homomorphism* from R to S if $R \subseteq \Phi\,S\,\Psi^{\mathsf{T}}$ holds. If, in addition, the pair $\mathcal{H}^{\mathsf{T}} = (\Phi^{\mathsf{T}}, \Psi^{\mathsf{T}})$ is a homomorphism from S to R, then $\mathcal{H}$ is said to be an *isomorphism* between R and S. Therefore, an isomorphism $\mathcal{I} = (\Phi, \Psi)$ between R and S is a pair of bijective mappings $\Phi : X_1 \leftrightarrow X_2$ and

$\Psi : Y_1 \leftrightarrow Y_2$, which satisfies the condition $R\,\Psi = \Phi\,S$. If R and S are homogeneous, then Φ is briefly called a homomorphism (isomorphism) if the pair (Φ, Φ) is a homomorphism (isomorphism).

2.2.3 Description of Sets

Relational algebra offers different ways of describing the subsets of a given set. In the following, we consider two representations.

The first representation uses *vectors*, i.e., relations $v : X \leftrightarrow Y$ with $v = v\,\mathsf{L}$. This condition means: Whatever set Z and universal relation $\mathsf{L} : Y \leftrightarrow Z$ we choose, an element x from X is either in relation $v\,\mathsf{L}$ to none of the elements of Z or to all elements of Z. As for a vector $v : X \leftrightarrow Y$ the range Y is irrelevant, we consider in the following almost only vectors $v : X \leftrightarrow \mathbf{1}$ with a specific singleton set $\mathbf{1}$ as range and omit the second subscript. Such a vector can be considered as a Boolean matrix with exactly one column, i.e., as a Boolean column vector, and describes the subset $\{x \in X : v_x\}$ of X. In the literature, for $R : X \leftrightarrow Y$ also the vector $R\,\mathsf{L} : X \leftrightarrow \mathbf{1}$ is called the *domain* of R.

A vector v is said to be a *point* if it is injective and surjective. For $v : X \leftrightarrow \mathbf{1}$ these properties mean that it describes a singleton set, i.e., an element of X. In the Boolean matrix model, hence a point is a Boolean column vector in which exactly one component is true.

Instead of vectors, we can use *injective embedding mappings* as a second way for representing subsets of a given set. Given an injective mapping $\imath : Y \leftrightarrow X$, we call Y a subset of X given by $\imath$. If Y is a subset of X given by $\imath$, then the vector $\imath^{\mathsf{T}}\,\mathsf{L} : X \leftrightarrow \mathbf{1}$, where $\mathsf{L} : Y \leftrightarrow \mathbf{1}$, describes Y in the above sense. Clearly, the transition in the other direction, i.e., the construction of an injective mapping $\mathsf{inj}(v) : Y \leftrightarrow X$ from a given vector $v : X \leftrightarrow \mathbf{1}$ describing Y, is also possible. In this case we have

$$(\mathrm{I}_1)\ \ \mathsf{inj}(v) \text{ is injective mapping} \qquad\qquad (\mathrm{I}_2)\ \ v = \mathsf{inj}(v)^{\mathsf{T}}\,\mathsf{L}\,.$$

It can easily be shown that these laws determine $\mathsf{inj}(v)$ up to isomorphism. Namely, if $v_1 : X_1 \leftrightarrow \mathbf{1}$ and $v_2 : X_2 \leftrightarrow \mathbf{1}$ are vectors describing a subset Y_1 of X_1 resp. Y_2 of X_2 and, furthermore, $\Psi : X_1 \leftrightarrow X_2$ is a bijective mapping, then $\mathcal{I} = (\Phi, \Psi)$, where $\Phi = \mathsf{inj}(v_1)\,\Psi\,\mathsf{inj}(v_2)^{\mathsf{T}}$ defines a bijective mapping $\Phi : Y_1 \leftrightarrow Y_2$, is an isomorphism between $\mathsf{inj}(v_1)$ and $\mathsf{inj}(v_2)$.

In combination with the set-theoretic membership relation (the relation-level equivalent of the meta-level symbol "$\in$") $\varepsilon : X \leftrightarrow 2^X$, defined by ε_{xs} if and only if $x \in s$, injective mappings can be used to enumerate sets of sets. More specifically, if the vector $v : 2^X \leftrightarrow \mathbf{1}$ describes a subset $\mathcal{S}$ of the powerset 2^X, then it is straightforward to compute an injection $\mathsf{inj}(v) : \mathcal{S} \leftrightarrow 2^X$, from which we obtain the elements of $\mathcal{S}$ as the columns of the relation $\varepsilon\,\mathsf{inj}(v)^{\mathsf{T}} : X \leftrightarrow \mathcal{S}$. If X is finite, this leads to an economic representation of $\mathcal{S}$ by a Boolean matrix with $|X|$ rows and $|\mathcal{S}|$ columns.

2.3 Some Specific Functions on Relations

In this subsection, we consider some special functions (in the everyday's sense) from relations to relations. Sometimes, they are also called *operations*. The functions we will present in the following are introduced in terms of the basic operations and, thus, in most cases they are only partially defined. As we will see later on, all functions easily can be computed using the RELVIEW system.

2.3.1 Closures

Let $R : X \leftrightarrow X$ be a homogeneous relation. The *reflexive closure* of R, i.e., the least reflexive relation containing R, simply computes to $R \cup \mathsf{I}$. The least transitive relation containing R is called the *transitive closure* of R and denoted by R^+, while the least reflexive and transitive relation containing R is called the *reflexive-transitive closure* of R and denoted by R^*. Using the fixed point theorems for monotone resp. $\cup$-continuous functions on complete lattices, we obtain the representations $R^+ = \bigcup_{i \geq 1} R^i$ and $R^* = \bigcup_{i \geq 0} R^i$. The transitive and reflexive-transitive closure are linked together by the equations $R^+ = R\,R^* = R^*\,R$ and $R^* = \mathsf{I} \cup R^+$.

2.3.2 Residuals and Symmetric Quotients

Residuals are the greatest solutions of certain inclusions. The *left residual* of S over R (in symbols $S \,/\, R$) is the greatest relation X such that $X\,R \subseteq S$ and the *right residual* of S over R (in symbols $R \backslash S$) is the greatest relation X such that $R\,X \subseteq S$. We will also need relations which are left and right residuals simultaneously, viz. *symmetric quotients*. The symmetric quotient $\mathsf{syq}(R, S)$ of two relations R and S is defined as the greatest relation X such that $R\,X \subseteq S$ and $X\,S^\mathsf{T} \subseteq R^\mathsf{T}$. In terms of the basic operations we have

$$S \,/\, R = \overline{\overline{S}\;R^\mathsf{T}} \qquad\qquad R \backslash S = \overline{R^\mathsf{T}\;\overline{S}}$$

as representations for the left residual resp. right residual and

$$\mathsf{syq}(R, S) = (R \backslash S) \cap (R^\mathsf{T} \,/\, S^\mathsf{T})$$

as representation for the symmetric quotient. The left residual is only defined if both relations have the same range and the right residual and the symmetric quotient are only defined if both relations have the same domain. Translating the two equations for the residuals into component-wise predicate logic notation yields

$$(S \,/\, R)_{yx} \Longleftrightarrow \forall\, z\;\, R_{xz} \rightarrow S_{yz} \qquad\qquad (R \backslash S)_{xy} \Longleftrightarrow \forall\, z\;\, R_{zx} \rightarrow S_{zy}\,.$$

In particular, for $S : Y \leftrightarrow Z$ and $R : Z \leftrightarrow X$, a universal relation $\mathsf{L} : \mathbf{1} \leftrightarrow Z$, and an empty vector $\mathsf{O} : Z \leftrightarrow \mathbf{1}$ we obtain the two correspondences

$$(S \,/\, \mathsf{L})_y \Longleftrightarrow \forall\, z\;\, S_{yz} \qquad\qquad (\,\overline{R} \backslash \mathsf{O})_x \Longleftrightarrow \forall\, z\;\, R_{zx}$$

for single first-order universal quantification. And, finally, in component-wise notation the symmetric quotient satisfies the equivalence

$$\mathsf{syq}(R, S)_{xy} \Longleftrightarrow \forall\, z\;\, R_{zx} \leftrightarrow S_{zy}\,.$$

If we consider this for the special case where R is a membership relation $\varepsilon : X \leftrightarrow 2^X$ and S is a vector $v : X \leftrightarrow \mathbf{1}$, then the type of $\mathsf{syq}(\varepsilon, v)$ is $[2^X \leftrightarrow \mathbf{1}]$ and for each set Y from 2^X we have $\mathsf{syq}(\varepsilon, v)_Y$ if and only if $\forall\, z\; z \in Y \leftrightarrow v_z$. Hence, $\mathsf{syq}(\varepsilon, v) : 2^X \leftrightarrow \mathbf{1}$ is exactly the point in the powerset corresponding to the vector v.

2.3.3 Choice Operations

In the Boolean matrix model of relations underlying the RELVIEW system the so-called point axiom [21] holds, saying that for every non-empty relation R there exist two points p and q such that $p\,q^{\mathsf{T}} \subseteq R$. In the special case of a non-empty vector $v : X \leftrightarrow \mathbf{1}$ from the point axiom we obtain the existence of a point $p : X \leftrightarrow \mathbf{1}$ contained in v. The choice of an element (expressed by a point p) from a non-empty vector (set) or of an ordered pair (expressed by the composition $p\,q^{\mathsf{T}}$ of points p, q) from a non-empty relation is fundamental for programming relational algorithms and, therefore, also included in the language of the RELVIEW system.

Our axiomatization of the *choice* $\mathsf{point}(v)$ *which selects an element from a non-empty vector* v is given by

$$(\mathrm{E}_1)\ \ \mathsf{point}(v) \subseteq v \qquad\qquad (\mathrm{E}_2)\ \ \mathsf{point}(v) \text{ is point}\,.$$

In the Boolean matrix model of relations, every relation containing exactly one ordered pair (x, y) is an atom in the lattice-theoretic sense. Therefore, we have decided to denote the *choice of an ordered pair from a non-empty relation* R by $\mathsf{atom}(R)$. The axioms which characterize this choice operation are

$$(\mathrm{A}_1)\ \ \mathsf{atom}(R) \subseteq R \qquad\qquad (\mathrm{A}_2)\ \ \mathsf{atom}(R)\,\mathsf{L} \text{ is point} \qquad\qquad (\mathrm{A}_3)\ \ \mathsf{atom}(R)^{\mathsf{T}}\,\mathsf{L} \text{ is point}\,.$$

In the Boolean matrix model of relations, the application $\mathsf{atom}(R)$ yields a Boolean matrix in which exactly one entry is true. Note that the types of v and $\mathsf{point}(v)$ as well as of R and $\mathsf{atom}(R)$ coincide.

2.3.4 Generation of Finite Carrier Sets

The RELVIEW system deals only with relations with finite domain and range. Hence, we are allowed to assume that every carrier set $X = \{x_1, \ldots, x_n\}$ of a relation of the workspace of RELVIEW is finitely generated by the specific (initial) element x_1 and a partial successor function mapping x_i to x_{i+1} for all i, $1 \leq i \leq n-1$. Like the choice of an element from a non-empty vector respectively an ordered pair from a non-empty relation, also the exhaustion of finitely generated carrier sets using an initial element and a partial successor operation is fundamental for programming relational algorithms. Therefore, corresponding constructions are included in the language of RELVIEW.

If we describe the initial element of X and the partial successor function on X in relation-algebraic terms, then this means that we have a point $\mathsf{init} : X \leftrightarrow \mathbf{1}$ and a relation $\mathsf{succ} : X \leftrightarrow X$ such that the properties

$$(\mathrm{G}_1)\ \ \mathsf{succ} \text{ is univalent, injective} \qquad\qquad (\mathrm{G}_2)\ \ \mathsf{succ}^{\mathsf{T}}\mathsf{L} \subseteq \overline{\mathsf{init}} \qquad\qquad (\mathrm{G}_3)\ \ (\mathsf{succ}^{\mathsf{T}})^{*}\,\mathsf{init} = \mathsf{L}$$

hold. The second formula says that the point init is not a successor. If we define a partial "next point function" by $\mathsf{next}(x) = \mathsf{succ}^{\mathsf{T}}\,x$, then the third axiom expresses the fact that every point $p : X \leftrightarrow \mathbf{1}$ can be obtained from init by finitely many applications of next. Our axiomatization of the pair $(\mathsf{init}, \mathsf{succ})$ of *initial point* and *univalent and injective successor relation* by (G_1) through (G_3) is a variant of the relational version of the well-known Peano axioms for natural numbers given in [2].

2.3.5 Truth Values and Tests

Using the only two relations $\mathsf{O} : \mathbf{1} \leftrightarrow \mathbf{1}$ and $\mathsf{L} : \mathbf{1} \leftrightarrow \mathbf{1}$ on the singleton set $\mathbf{1}$ as the truth values (Booleans), it is even possible to test properties of relations. The most important test is relational inclusion $R \subseteq S$ which is, for $R : X \leftrightarrow Y$ and $S : X \leftrightarrow Y$ of the same type, defined by the first-order formula $\forall\, x, y\ R_{xy} \rightarrow S_{xy}$. By the rules of universal quantification and the propositional fact that $p \rightarrow q$ is equivalent to $\neg p \vee q$ we get that $R \subseteq S$ if and only if $((R \cap \overline{S}\,)\,\mathsf{L}) \setminus \mathsf{O} = \mathsf{L}$ with vectors $\mathsf{O} : X \leftrightarrow \mathbf{1}$ and $\mathsf{L} : Y \leftrightarrow \mathbf{1}$. See also [5].

Therefore, we can integrate the test on inclusion as an operation incl, where the truth value $\mathsf{incl}(R, S) : \mathbf{1} \leftrightarrow \mathbf{1}$ is defined by

$$\mathsf{incl}(R, S) = ((R \cap \overline{S}\,)\,\mathsf{L}) \setminus \mathsf{O}\,.$$

Further relational properties consisting of inclusions, such as equality, univalencity, totality, surjectivity, injectivity, transitivity, reflexivity, antisymmetry, and many others, can then easily be reformulated in terms of incl since, clearly, the propositional connectives directly correspond to the Boolean operations on relations.

2.4 Relational Domain Constructions

Domains are used, for instance, in denotational semantics or mathematical logic to interpret types, and usually constructed step be step starting from primitive domains. Such constructions can also be described with relational means. In the following, we describe some important domain constructions which also are implemented in the RELVIEW system. Note that these constructions may or may not exist in an arbitrary model of abstract relational algebra. However, this problem does not occur in the case of the Boolean matrix model of concrete relations underlying RELVIEW.

2.4.1 Binary Direct Product

Within the framework of abstract relational algebra it is natural to characterize direct products by means of the natural projections, see [21, 28]. Then one obtains the following specification: We call a pair $\pi_i : PX \leftrightarrow X_i$, $1 \le i \le 2$, a (binary) *direct product* if

$$\begin{array}{llll} (\mathrm{P}_1) & {\pi_1}^{\mathsf{T}}\, \pi_1 = \mathsf{I} & (\mathrm{P}_2) & {\pi_2}^{\mathsf{T}}\, \pi_2 = \mathsf{I} \\ (\mathrm{P}_3) & \pi_1\, {\pi_1}^{\mathsf{T}} \cap \pi_2\, {\pi_2}^{\mathsf{T}} = \mathsf{I} & (\mathrm{P}_4) & {\pi_1}^{\mathsf{T}}\, \pi_2 = \mathsf{L}\,. \end{array}$$

It is easy to verify that the natural projections from a Cartesian product $X_1 \times X_2$ to the components X_i are a model of (P_1) through (P_4) if the placeholder PX is replaced by $X_1 \times X_2$. By purely relation-algebraic reasoning, furthermore, it can be shown that the direct product is uniquely characterized up to isomorphism: Let $\rho_i : PY \leftrightarrow Y_i$, $1 \le i \le 2$, be another model of the above axioms and assume a pair $\Psi_i : X_i \leftrightarrow Y_i$, $1 \le i \le 2$, of bijective mappings. Then, for each $i, 1 \le i \le 2$, we can establish an isomorphism between π_i and ρ_i by the pair $\mathcal{I}_i = (\Phi, \Psi_i)$, where $\Phi = \pi_1\, \Psi_1\, {\rho_1}^{\mathsf{T}} \cap \pi_2\, \Psi_2\, {\rho_2}^{\mathsf{T}}$ defines a bijective mapping $\Phi : PX \leftrightarrow PY$.

Based on two binary direct products (π_1, π_2) and (ρ_1, ρ_2) we define the following two operations, called *tupling* (or *fork*) resp. *parallel composition*:

$$[R, S] = R\,\pi_1^{\mathsf{T}} \cap S\,\pi_2^{\mathsf{T}} \qquad\qquad R \,\|\, S = \pi_1\,\Psi_1\,\rho_1^{\mathsf{T}} \cap \pi_2\,\Psi_2\,\rho_2^{\mathsf{T}}\,.$$

If R and S are partial orderings, then also their parallel composition $R \,\|\, S$ is a partial ordering, called *product ordering*.

2.4.2 Binary Direct Sum

The direct sum (or disjoint union) can be defined in largely the same fashion as the direct product. Dually to the natural projections the natural injections are used, see [28]. Then one obtains the following specification: We call a pair $\imath_i : X_i \leftrightarrow SX$, $1 \leq i \leq 2$, a (binary) *direct sum* if

$$(\mathrm{S}_1)\;\; \imath_1\,\imath_1^{\mathsf{T}} = \mathsf{I} \qquad\qquad (\mathrm{S}_2)\;\; \imath_2\,\imath_2^{\mathsf{T}} = \mathsf{I}$$
$$(\mathrm{S}_3)\;\; \imath_1^{\mathsf{T}}\,\imath_1 \cup \imath_2^{\mathsf{T}}\,\imath_1 = \mathsf{I} \qquad\qquad (\mathrm{S}_4)\;\; \imath_1\,\imath_2^{\mathsf{T}} = \mathsf{O}\,.$$

Given sets X_i, $1 \leq i \leq 2$, it is easy to verify that the injections from these sets to the direct sum $X_1 + X_2$ (replacing the placeholder SX) are a model of (S_1) through (S_4). Again by purely relation-algebraic reasoning it can be shown that by the laws the direct sum is uniquely characterized up to isomorphism. Namely, if $\kappa_i : Y_i \leftrightarrow SY$, $1 \leq i \leq 2$, is another direct sum and we have two bijective mappings $\Phi_i : X_i \leftrightarrow Y_i$, $1 \leq i \leq 2$, then for each $i, 1 \leq i \leq 2$, the pair $\mathcal{I}_i = (\Phi_i, \Psi)$ is an isomorphism between $\imath_i$ and κ_i, where $\Psi = \imath_1^{\mathsf{T}}\,\Phi_1\,\kappa_1 \cup \imath_2^{\mathsf{T}}\,\Phi_2\,\kappa_2$ defines again a bijective mapping $\Psi : SX \leftrightarrow SY$.

Dually to tupling and parallel composition, we have in the case of two direct sums $(\imath_1, \imath_2)$ and (κ_1, κ_2) the following operations:

$$R + S = \imath_1^{\mathsf{T}}\,R \cup \imath_2^{\mathsf{T}}\,S \qquad\qquad \mathsf{B}(R, S) = \imath_1^{\mathsf{T}}\,R\,\kappa_1 \cup \imath_2^{\mathsf{T}}\,S\,\kappa_2\,.$$

Since direct sums are not used as much as direct products, in the literature one finds not fixed names for these operations. We call $R + S$ the *relational sum* of R and S. If R and S are partial orderings, then also $\mathsf{B}(R, S)$ is a partial ordering, which we call *sum ordering*. Our notation using B comes from the fact, that for a bipartite graph $\mathcal{B} = (X, Y, R, S)$ with relations $R : X \leftrightarrow Y$ and $S : Y \leftrightarrow X$ an application of B yields the corresponding "ordinary" graph $\mathcal{G} = (V, B)$ with node set $V = X + Y$ and homogeneous relation $B = \mathsf{B}(R, S)$ on V.

2.4.3 Membership and Powersets

A relation-algebraic characterization of the powerset 2^X of a set X can conveniently be done using the set-theoretic membership relation induced in the section on the description of sets. Formally, we call a relation $\varepsilon : X \leftrightarrow PX$ a *powerset relation* if

$$(\mathrm{M}_1)\;\; \mathsf{syq}(\varepsilon, \varepsilon) \subseteq \mathsf{I} \qquad\qquad (\mathrm{M}_2)\;\; \forall\, R\; (\mathsf{L}\,\mathsf{syq}(\varepsilon, R) = \mathsf{L})\,.$$

In the concrete case of a membership relation $\varepsilon : X \leftrightarrow 2^X$ the first axiom corresponds to the extensionality axiom saying that sets are equal if and only if they contain the same elements, whereas the second axiom corresponds to the set comprehension principle since it says that every vector $v : X \leftrightarrow \mathbf{1}$ (representing a subset of X) has a corresponding point $\mathsf{syq}(\varepsilon, v) : 2^X \leftrightarrow \mathbf{1}$ (i.e., an element) in the powerset. This shows that the usual membership relation is a powerset relation. The function $v \mapsto \mathsf{syq}(\varepsilon, v)$ is injective and its left-inverse on points is $p \mapsto \varepsilon\, p$. Hence, these functions establish some kind of isomorphism between subsets of X and elements of 2^X.

Since every relation-algebraic equation using ε is translated into a formula with higher-order quantification, in axiom (M_2) the higher-order quantification (over relations) does not surprise. Again it can be shown by purely relation-algebraic reasoning that the powerset relation is uniquely characterized up to isomorphism. Indeed, if $\varepsilon' : Y \leftrightarrow PY$ is another powerset relation, $\Phi : X \leftrightarrow Y$ is a bijective mapping, and one defines the bijective mapping $\Psi : PX \leftrightarrow PY$ by $\Psi = \mathsf{syq}(\varepsilon, \Phi\, \varepsilon')$, then $\mathcal{I} = (\Phi, \Psi)$ is an isomorphism between ε and ε'.

2.4.4 Domains of Partial and Total Mappings

We consider the set Y^X of the partial mappings from X to Y to be a subset of the set $2^{X \times Y}$ of all relations from X to Y. Let $\pi_1 : X \times Y \leftrightarrow X$ and $\pi_2 : X \times Y \leftrightarrow Y$ be the projections from $X \times Y$ to X and Y, respectively. We demand the point $\mathsf{syq}(\varepsilon, v) : 2^{X \times Y} \leftrightarrow \mathbf{1}$ to be contained in Y^X if and only if the vector $v : X \times Y \leftrightarrow \mathbf{1}$ describes a partial mapping as subset of $X \times Y$ in the usual set-theoretic sense.

Using a relation-algebraic notation, the usual set-theoretic definition that $(x_1, y_1) \in v$ and $(x_2, y_2) \in v$ and $\pi_1(x_1, y_1) = \pi_1(x_2, y_2)$ implies $\pi_2(x_1, y_1) = \pi_2(x_2, y_2)$ for all pairs (x_1, y_1) and (x_2, y_2) becomes the inclusion $v\, v^\mathsf{T} \cap \pi_1 \pi_1{}^\mathsf{T} \subseteq \pi_2 \pi_2{}^\mathsf{T}$. This leads to the following axiomatization of the domain $(\pi_1, \pi_2, \varepsilon_F)$ of the *partial mappings* which is a refinement of the above axioms of the powerset.

(F_1) (π_1, π_2) is direct product in the sense of 2.4.1

(F_2) $\mathsf{syq}(\varepsilon_F, \varepsilon_F) \subseteq \mathsf{I}$

(F_3) $\forall\, R\ (\mathsf{L}\, \mathsf{syq}(\varepsilon_F, R) = \mathsf{L} \leftrightarrow R\, R^\mathsf{T} \cap \pi_1 \pi_1{}^\mathsf{T} \subseteq \pi_2 \pi_2{}^\mathsf{T})$.

Note that in the set-theoretic standard model of partial mappings $\varepsilon_F : X \times Y \leftrightarrow Y^X$ is a membership relation.

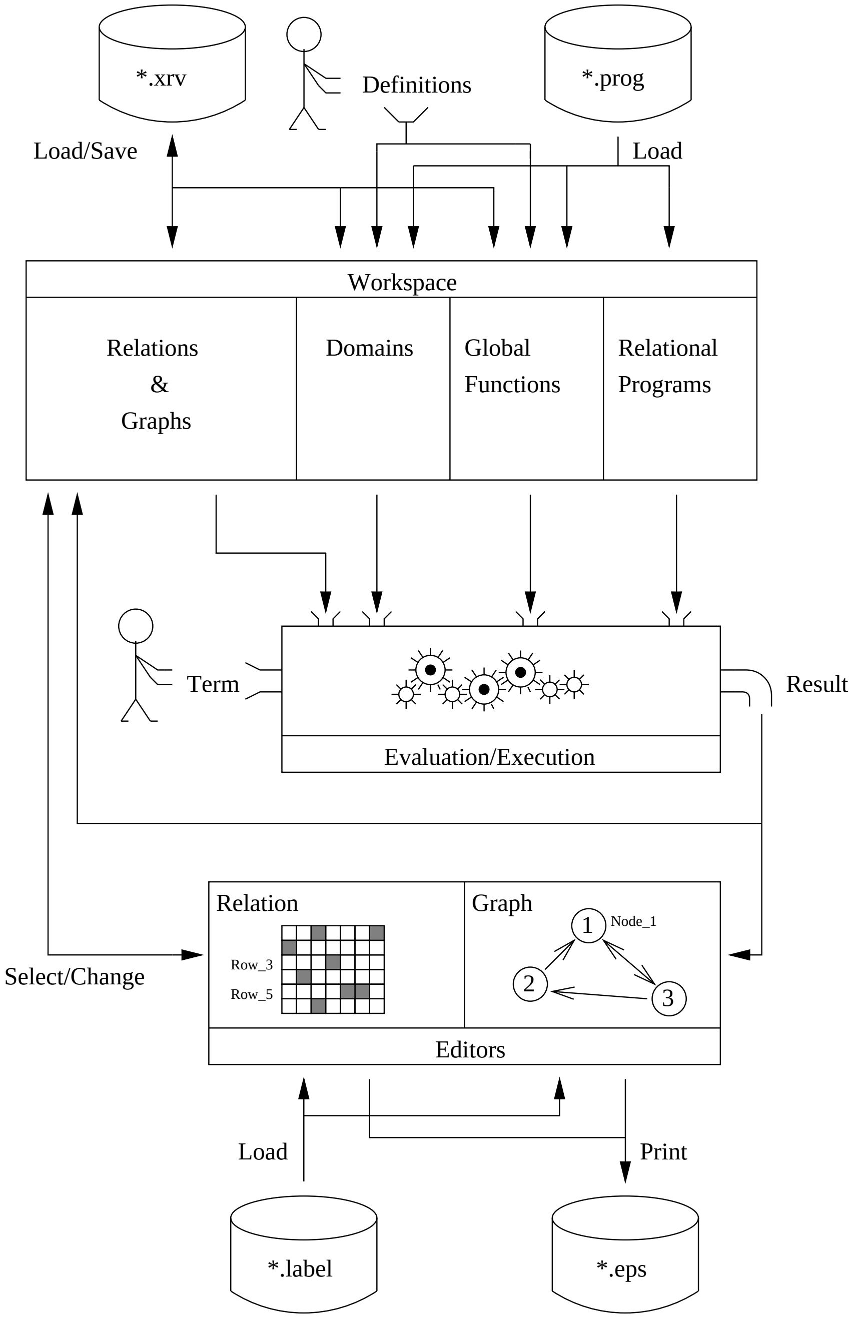
The equation $v^\mathsf{T} \pi_1 = \mathsf{L}$ expresses the fact that the vector $v : X \times Y \leftrightarrow \mathbf{1}$ describes a subset of $X \times Y$ which is a total relation in the set-theoretic sense. Hence, to characterize the domain of total mappings by a triple $(\pi_1, \pi_2, \varepsilon_F)$ only the right-hand side of the equivalence in (F_3) must be completed by "$\wedge\ R^\mathsf{T} \pi_1 = \mathsf{L}$". For details, see [1, 28].

3 The RELVIEW System

After giving a short overview of the structure of the system, we describe the graphical user interface in the Sections 3.2 to 3.11. The syntax of relational identifiers, expressions, functions, and programs is presented in the Sections 3.12 to 3.17. Section 3.18 deals with the labeling mechanism and Section 3.19 gives some configuration and installation hints.

3.1 General Structure of the System

The general structure of the RELVIEW system can be illustrated as follows:



Roughly spoken, the system consists of three components: The "workspace" holds relations and graphs, definitions of relational domains, global functions, and relational programs. The second component is the evaluation resp. execution unit. A relational term entered by the user is evaluated based on the objects contained in the workspace. The result, a relation, is written back to the workspace again. Here we want to remark that the workspace always contains a relation with name "$" which not only can act as an argument in calculations but also denotes the result of an evaluation if no other relation name is given. Evaluation results, i.e., relations, are also handed over to the relation editor which is part of the third component of the system. The two editors of this component are not only used to display relations and graphs but also form the base for entering these two kind of objects into the system. At this point we want to mention that in the context of RELVIEW a "relation" is always a Boolean matrix. Therefore, in the following we will speak of rows, columns, and dimension of a relation.

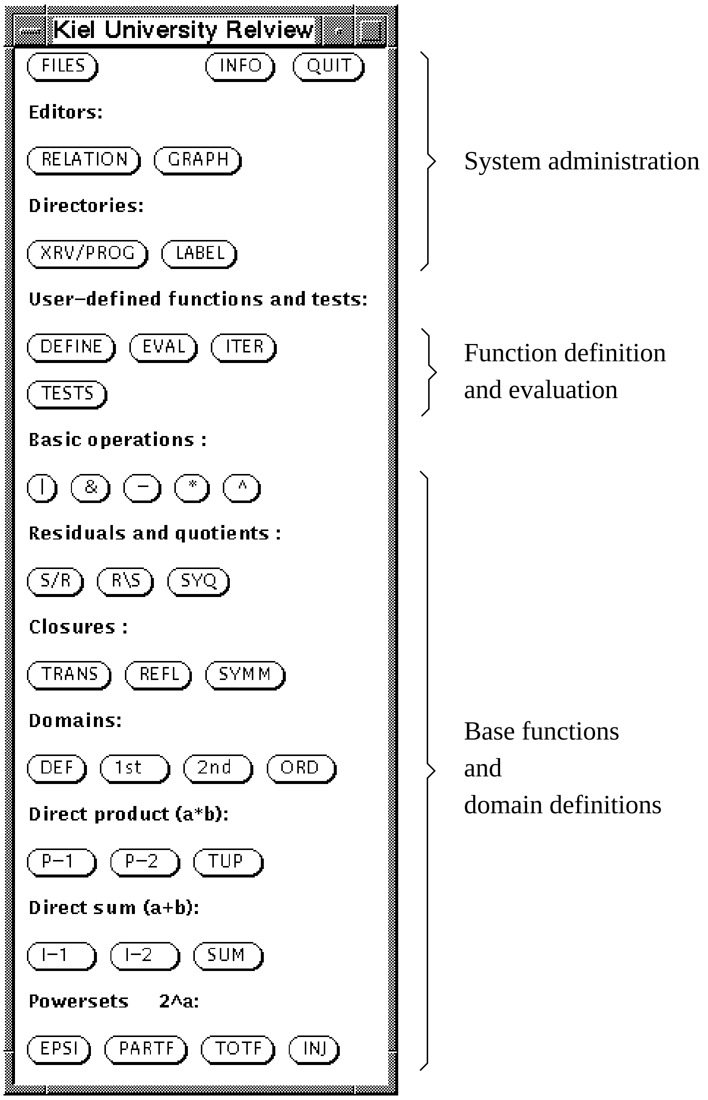
As the picture shows, four different types of files are supported by the system. Files with the extension ".xrv" or xrv-files for short can hold relations, graphs, and definitions of relational domains and global functions. These files are not human-readable and created by the system itself. The second type of files have the extension ".prog" and are called program files or prog-files for short. Program files can hold definitions of relational domains, global functions, and relational programs. These files are human-readable, i.e., they can be created using a text-editor. Note that program files can be loaded into the system but not written back to disk. The third kind of files involved are files with the extension ".label", label files for short. Label files contain definitions of label sets which can be used for labeling rows and/or columns of relations and nodes of graphs for illustration purposes. As in the case of program files, label files are ordinary text files, i.e., they are human-readable, and can only be loaded into the system. The last type of files are encapsulated postscript-files which are created by the system and contain relations and graphs in a printable format. These exported drawings can be, e.g., included in LaTeX-documents.

Please note that beside the described user interactions – entering relations and graphs and entering relational terms for evaluation and loading and saving various kinds of files – definitions of relational domains and global functions can be put into the system using special dialog windows which are not explicitly shown in the above picture.

In the following sections we will describe the various windows of the graphical user interface of RELVIEW in greater detail.

3.2 The Menu Window

After starting the system the menu window is presented to the user. Conceptually the menu window can be devided into three parts. The first three button rows deal with system administration tasks like opening the file-chooser, the information window, the two directory windows and the windows of the two editors, and quitting the system. The detailed appeareance of this window strongly depends on various resources defined in the configuration file, see Section 3.19.1. Typically it looks as follows:



In detail the following actions are invoked by the different buttons in the first three rows of the menu window:

FILES	:	Opens the file-chooser window (see Section 3.11).
INFO	:	Pops up an information window, presently showing the version number and a copyright notice only.
QUIT	:	Quits the system.
RELATION	:	Opens the window of the relation editor (see Section 3.4).
GRAPH	:	Pops up the window of the graph editor (see Section 3.5).
XRV/PROG	:	Displays the directory window showing the state of the workspace (see Section 3.3).
LABEL	:	Opens the label directory listing label sets which are loaded into the system (see Section 3.18).

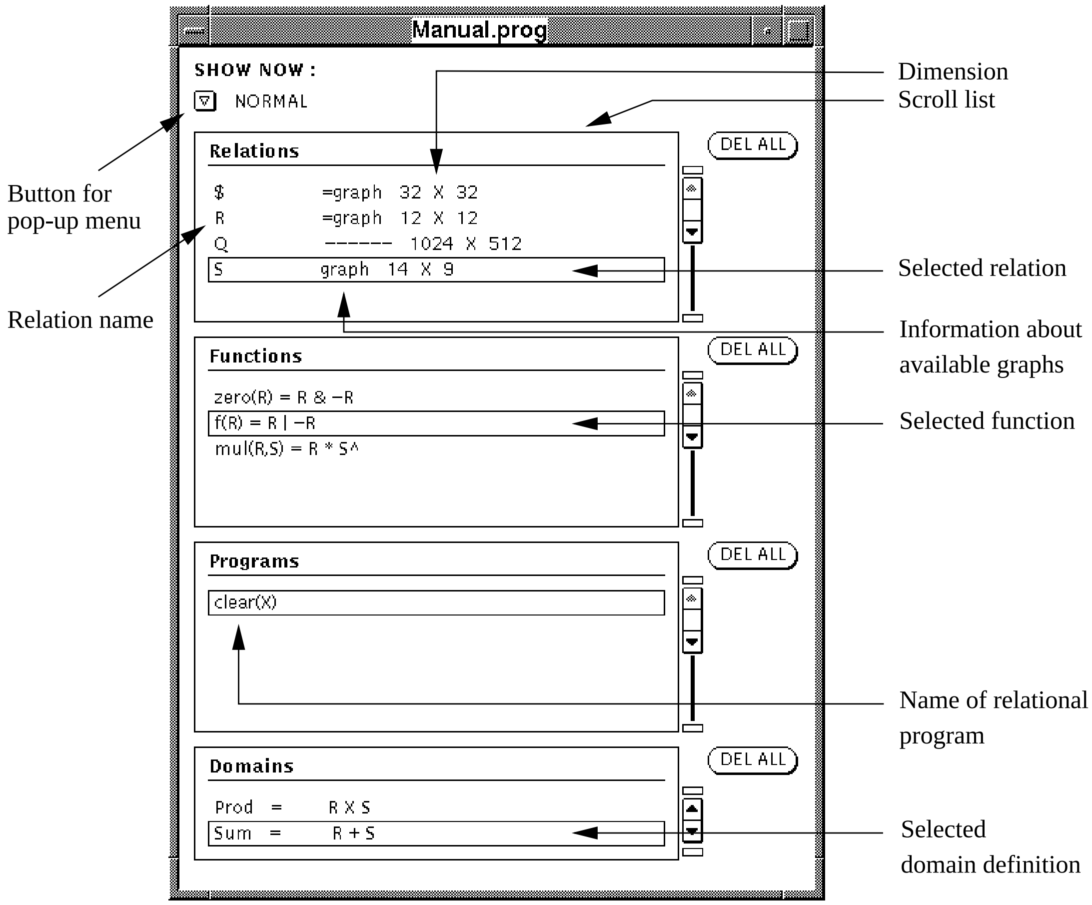
The buttons in the second part of the menu window cover these kind of actions which are mostly needed while working with the system:

DEFINE : Opens a dialog window for entering a definition of a global function (see Section 3.6).
EVAL : Pops up the evaluation window for entering a relational term. Relational terms are described in Section 3.15.
ITER : Opens a window for iterated application of a function to a relation.
TESTS : Pops up a window for invoking relational tests (see Section 3.10).

Finally, in the third and last part of the menu window, a number of relational operations are directly accessible via push buttons. Additionally, in the part "Domains" a button "DEF" can be found which allows to enter definitions of relational domains into the system.

3.3 The Directory Window

The directory window presents the actual state of the workspace to the user. It contains four scroll lists showing the names and dimensions of relations and possibly existing graphs, the globally defined functions, loaded relational programs, and, finally, defined relational domains. Although the detailed appeareance depends on resources defined in the configuration file, typically the directory window looks as follows:



On the right hand side of each scroll list there is a button "DEL ALL" which allows to delete all relations/graphs, global functions, relational programs, and domain definitions, respectively, from the workspace. Please note that the relation "$" cannot be deleted from the workspace. The button below the text “SHOW NOW” pops up a menu containing the two entries “NORMAL” and “HIDDEN”. Relations, graphs, global functions, and domain definitions can be declared as hidden with the effect that these objects are not listed in the “normal” scroll lists but are shown in the “hidden” ones. Normally only relations, graphs, function and domain definitions contained in the startup-file are loaded with the attribute hidden into the system at startup-time. See Section 3.19.2 for a description of startup-files.

The first scroll list shows all names of relations and graphs stored in the workspace. An entry in this list can be of one of three following forms:

1. ≫Name≪ - - - - - - ≫Rows≪ X ≫Columns≪

   An entry of this form means that in the workspace there exists a relation with name “Name” and dimension “Rows” $\times$ “Columns”, i.e., with “Rows” rows and “Columns” columns.

2. ≫Name≪ =graph ≫Rows≪ X ≫Columns≪

   As in the previous case there exists a relation with name “Name” with dimension “Rows” $\times$ “Columns” in the workspace. In addition, there exists a graph with name “Name” which is in a set-theoretic sense equal to the relation, i.e., the graph and the relation describe the same mathematical object. Note that in this case the relation is a homogeneous one and the numbers of rows and columns coincide with the number of nodes in the graph.

3. ≫Name≪ graph ≫Rows≪ X ≫Columns≪

   There exists a relation with name “Name” with the given dimension and a graph with the same name “Name”. In contrast to the previous case, **it is not guaranteed that the relation and the graph describe the same mathematical object, i.e., the relation and the graph may differ**. This could have been happened by modifying the relation or the graph with one of the editors, creating the relation and the graph separately, renaming the relation, or by evaluating a relational term using the name “Name” for the resulting relation.

Clicking onto a list item with the left mouse button, selects the corresponding relation and/or graph. The selected relation is “loaded” into the relation editor and the possibly existing graph is drawn into the window of the graph editor. Refer to Sections 3.4 and 3.5 for a description of the two editors available in the system.

The second scroll list shows all global functions existing in the workspace. Beside the name and the parameters the defining term is also displayed. Clicking onto a list item with the left mouse button selects a function with the effect that the definition of that function is shown in the FUNCTION-DEFINITION window which allows editing and deleting. See Section 3.6 for a description of that dialog window and Section 3.16 for an introduction into global functions.

The third scroll list lists all loaded relational programs. It only presents the name and the formal parameters. Note that the body of a program is not shown. The program text can only be inspected and modified by editing the corresponding program file using a text-editor and subsequent reloading.

The last scroll list presents all defined relational domains with their names and definitions. Clicking onto a list item with the left mouse button selects a domain definition and displays it in the DOMAIN-DEFINITION window allowing editing and deleting. See Section 3.9 for a description of that window.

3.4 The Relation Editor

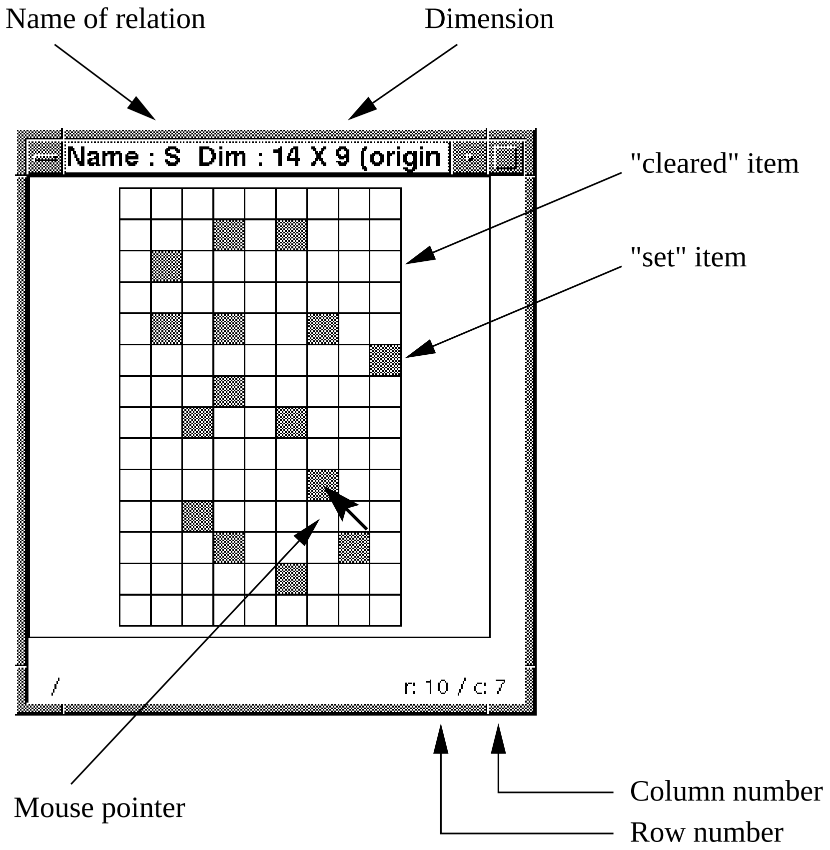
This section gives an introduction into the relation editor of RELVIEW. For simplicity we will make use of the following abbreviations:

LMB	:	left mouse button
MMB	:	middle mouse button
RMB	:	right mouse button

The window of the relation editor can be opened by clicking onto the button "RELATION" in the menu window. Selecting a relation in the first scroll list of the directory window loads this relation into the editor (see Section 3.3). Evaluation of relational terms performs a load command implicitly, i.e., the result of the evaluation, a relation, is automatically displayed in the relation editor window.

Please note that there is no explicit save command. Modifying the relation actually shown in the relation editor directly changes the relation stored in the workspace.

Typically the window of the relation editor looks as follows:



In the following by an "item" we mean a single entry of a relation unequivocal defined by a row and a column of a relation. In the relation editor such an item is graphically represented by a square as shown in the above picture.

An item is "set" if it describes a "true-entry" of a relation. The corresponding square of the graphic representation is a grey one. A "false-entry" of a relation is represented by a white square. In this case we speek of a "cleared" item.

A "line" can be a complete row or column or diagonal of a relation. The kind of line modifiable with the editor can be determined by selecting a menu entry; for details see below. We speek of a "completely set" line, if all items of a line are set and of a "completely cleared" or "empty" line, if all items are cleared.

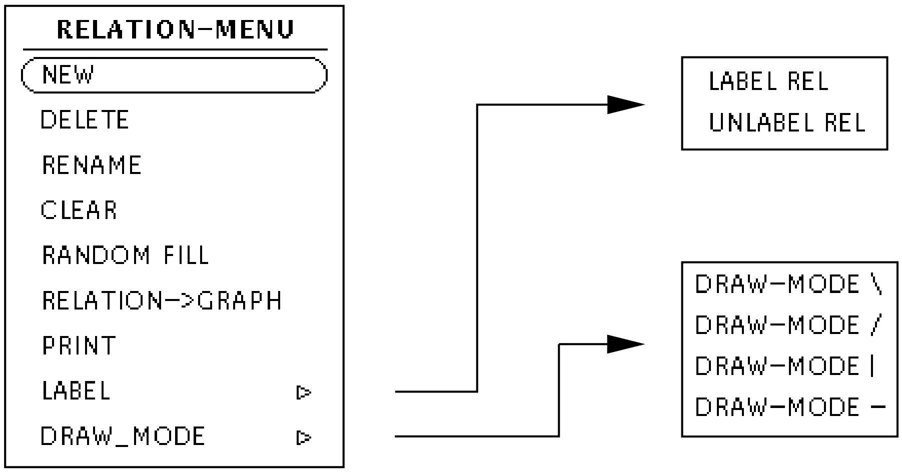
3.4.1 Actions Invoked by Mouse Buttons

If the mouse pointer is located on an item of a relation, the mouse buttons invoke the following different actions:

LMB	:	If the item is cleared, it will be set If the item is set, it will be cleared.
MMB	:	Set the line determined by the mouse position completely, if it is not set completely (e.g. empty). If the line choosen by the mouse pointer is set completely, the line is cleared completely.
RMB	:	Pops up a menu. All actions within the menu are selected **with the same, i.e, right mouse button.**

3.4.2 Pop Up Menu of the Relation Editor

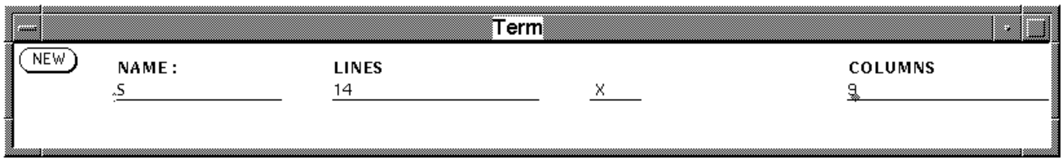
By pressing the RMB in the relation editor the following menus are reachable:



Selecting the various menu entries with the RMB invoke the following actions:

- NEW:

Opens the following dialog window which allows to define a new relation. The name and the number of rows and columns have to be entered into the different input fields:



The creation of the new relation can be invoked by pressing the button "NEW" or, alternatively, by pressing the "RETURN"-key on the keyboard. The "RETURN" or "TAB"-key can be used to switch over to the respective next input field. If there exists a relation with the newly choosen name in the workspace, the system deletes it after asking the user for confirmation. An existing graph with that name is bound to the newly created relation.

- DELETE:

  Deletes the relation displayed in the relation editor window from the workspace. **If there exists a graph with the same name as the relation, the graph is also deleted.** Before deleting the relation and a possibly existing graph, the system asks for confirmation by presenting a notice prompt to the user. In the case that the graph is displayed in the graph editor window, it can be restored by choosing the menu entry "GRAPH → RELATION" in the graph editor pop-up menu. See Section 3.5 for details. Note that the special relation "`$`" cannot be deleted.

- RENAME:

  Pops up a window for entering a new name for the relation visible in the relation editor window. Let `R` be the name of the relation in the editor window and let `Q` be the newly choosen name. Two cases can be distinguished.

  1. There exists a graph `R`: A relation `Q` and a graph `Q` possibly contained in the workspace are deleted. Here the system asks the user for confirmation. The relation `R` **and the graph** `R` are both renamed into `Q`.
  2. There exists no graph `R`: A relation `Q` contained in the workspace is deleted after asking the user for agreement. The relation `R` is renamed into `Q` and a possibly existing graph `Q` is bound to the "new" relation `Q`.

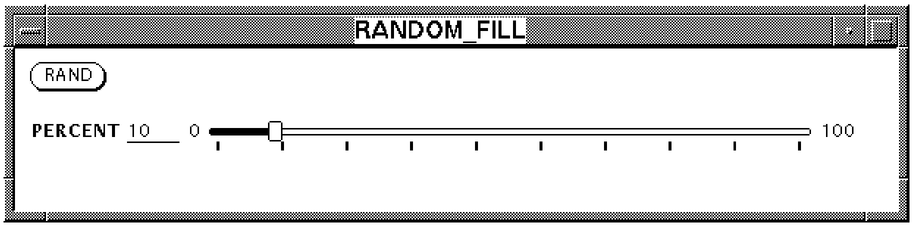
  In both cases, the renamed relation is displayed in the relation editor window. A graph which belongs to the renamed relation is shown in the graph editor window, see Section 3.5. Note that the special relation "`$`" cannot be renamed.

- CLEAR:

  Clears the whole relation, i.e., all items are cleared.

- RANDOM FILL:

  Opens the following window for entering a factor for filling the relation randomly:



Before filling the relation it is cleared completely.

- RELATION $\rightarrow$ GRAPH:

  Creates a graph from a homogeneous relation with the same name as the relation. The graph is displayed in the graph editor window; see Section 3.5. If no graph with the name of the relation exists in the workspace yet, then the nodes of the new graph are automatically placed on a circle and the arcs are drawn as lines. Otherwise, the RELVIEW system asks for confirmation before overwriting the existing graph. In the case that the number of rows and columns coincide with the number of nodes of the graph, the positions of the nodes are not changed. In the other case, the above mentioned standard placement is used again. The entry in the directory window belonging to the relation is updated, i.e., it shows the text "=graph" in the middle section.

- PRINT:

  Outputs the relation as an encapsulated postscript-file. As filename the name of the relation extended by ".rel.eps" is used. The file is created in the directory where the system has been started from.

- LABEL:

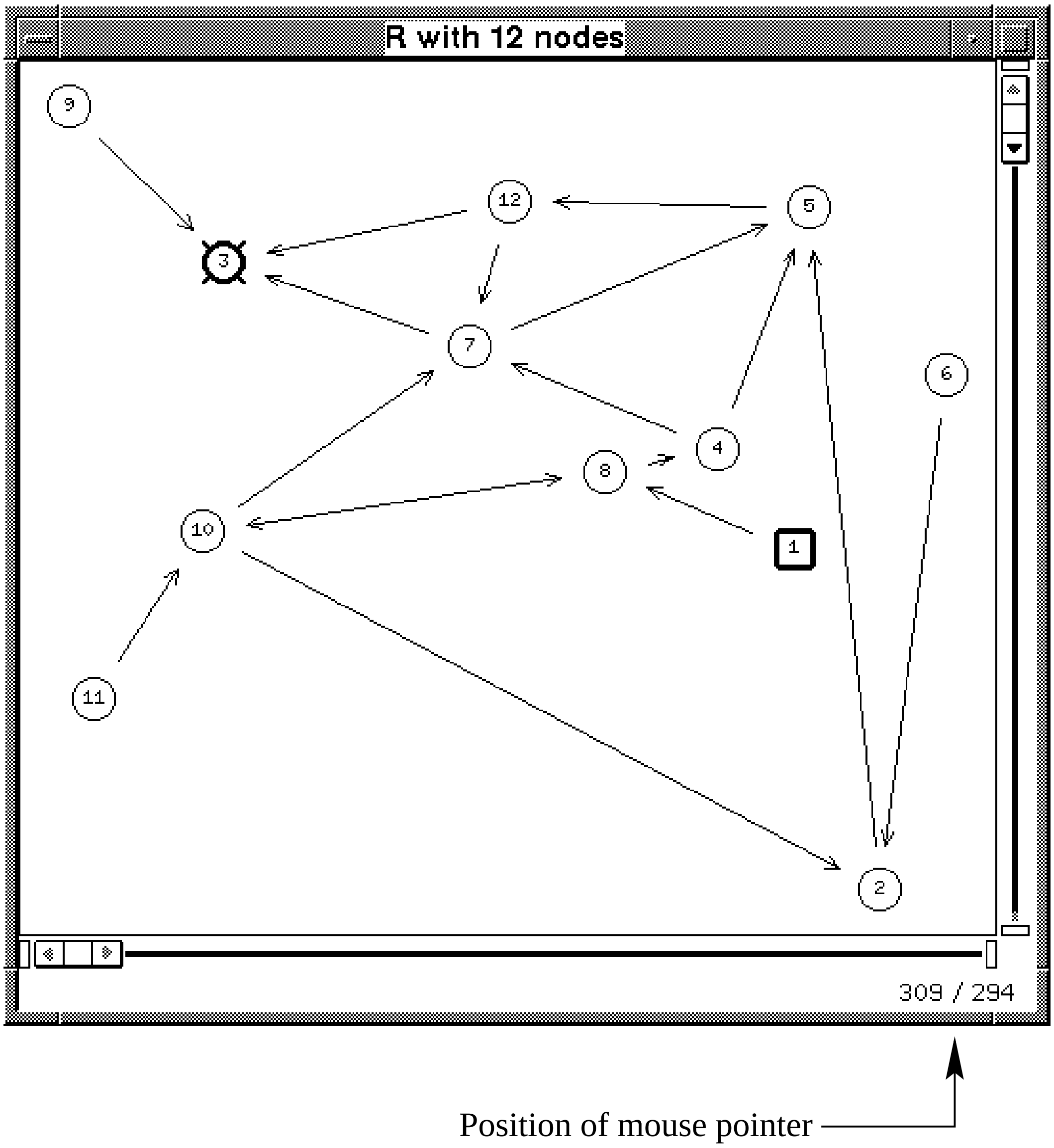
  Pops up a submenu as it is shown in the above picture which allows to add/remove labels to the rows and/or columns of the relation. Refer to Section 3.18 for a description of the labeling mechanisms of RELVIEW.

- DRAW_MODE:

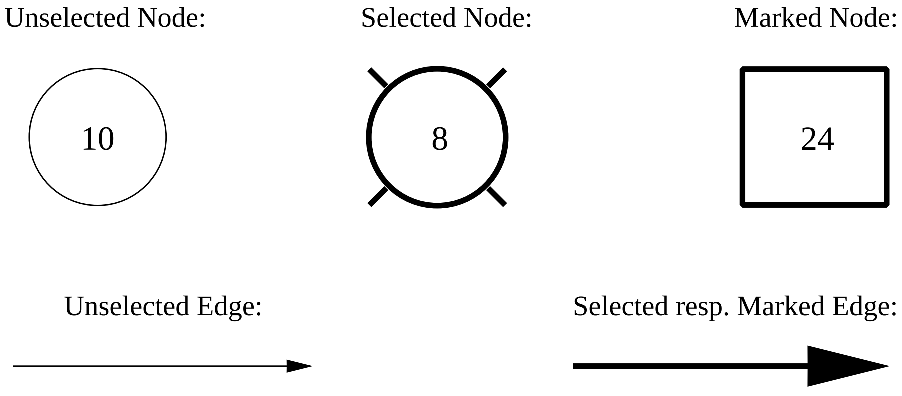
  Opens a submenu as presented above for selecting a draw mode for lines. Then four different modes can be choosen, viz. horizontal, vertical and two types of diagonal lines.

3.5 The Graph Editor

This section describes the graph editor of RELVIEW. The window of the graph editor can be opened by pressing the button "GRAPH" in the menu window. A typical view of the graph editor window can look as follows:



In the graph editor various kinds of nodes and edges are used:



As the picture illustrates, we speek of selected, unselected, and marked nodes and edges. Selecting a node allows moving and deleting it. In a similar way, a selected edge can be deleted. At one time, only one node or one edge can be selected. Marking nodes and edges can be used to illustrate computation results. All nodes of the graph are numerated by the system automatically. If there exists a relation which coincides with the graph, the numbering of the nodes corresponds to the numbering of the rows resp. columns of that relation. The allocated numbers always form an intervall.

Each node is surrounded by a "selection area". Clicking with the mouse into this area selects the corresponding node and performs a special action with this node. In this case we use the term that the mouse pointer is "nearby" a node. If the mouse pointer does not point into a selection area, we say that the mouse is "not nearby" a node.

As in the previous section, we make use of the abbreviations LMB, MMB, and RMB denoting the left, middle, and right mouse button, respectively.

3.5.1 Actions Invoked by Mouse Buttons

The different mouse buttons invoke the following actions depending on the state of nodes and egdes and the mouse position:

- Actions initiated by pressing the LMB:

  If the mouse pointer is **not nearby a node**:

  - If **no node is selected**, then a **new** node will be placed at the pointer position. The new node gets the smallest number which has not been assigned to an other node yet.
  - If **a node is selected**, then the selected node will be **moved** to the position of the pointer.

  If the mouse pointer is **nearby a node** N:

  - If **no node is selected**, then the node N nearby the pointer **is selected**.
  - If **a node** M **is selected**:
    - If **no egde exists** from the selected node M to the node N nearby the pointer position, then a **new egde is created** from the selected node M to the node N nearby the pointer.
    - If there **exists an edge** from the selected node M to the node N nearby the pointer, then the **egde will be selected**.

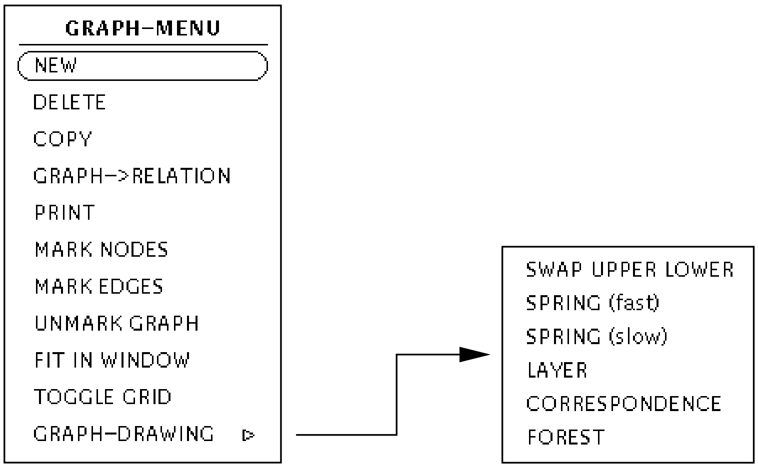
- Pressing the MMB **deletes** a selected node together with all edges connected to that node, a selected edge or the last created edge. In case of deleting a node, the remaining nodes are in general renumbered so that all numbers form an intervall again.

  Please be careful: There is no undo-function!

- The RMB pops up a menu. All actions within the menu are selected **with the same, i.e., right mouse button**.

3.5.2 Pop Up Menu of the Graph Editor

By pressing the RMB in the graph editor the following menus are reachable:



Selecting the various menu entries with the RMB invoke the following actions:

- NEW:

  Opens a dialog window which allows to enter a name for a new graph. Initially the new graph has no nodes. Let R be the name for the new graph. Three cases can be distinguished.

  1. There exists no relation R in the workspace: After drawing a graph with at least one node, a relation with the same name R can be created from this graph by choosing the menu entry GRAPH $\rightarrow$ RELATION as described below.

     Please note: If no relation is created, the newly drawn graph gets lost if an other graph contained in the workspace is loaded into the graph editor.

  2. The workspace contains a relation R but no graph R: After drawing a graph with at least one node, the new graph is bound to the relation R.
  3. There exists a relation R and a graph R: The existing graph is deleted. Before deleting the graph, the system asks the user for confirmation. After drawing a graph with at least one node, this new graph is bound to the relation R.

- DELETE:

  After asking the user for confirmation, all nodes of the graph are deleted.

- Copy:

  Opens a dialog window for entering a new name. Let R be the name of the graph in the editor window and let Q be the newly choosen name. If there exists a relation R, a copy with name Q of the graph R is created. Otherwise the graph R is renamed into Q. The "new" graph Q is displayed in the graph editor window. Depending on the contents of the workspace, several further actions are performed:

1. If the workspace contains a relation Q but no graph Q, the "new" graph Q is bound to the relation Q.
2. If there exist a relation Q and a graph Q, the graph Q is deleted after asking the user for agreement and the "new" graph Q is bound to the relation Q.
3. In the case that the workspace does not contain a relation with name Q, nothing more is done. By choosing the menu entry GRAPH → RELATION, a relation can be created from the "new" graph Q.

- GRAPH → RELATION:

  Creates a relation from the graph with the same name or updates an existing relation with the same name like the graph. In the first scroll list of the directory window, an appropriate list entry is created resp. updated. In particular, it shows the entry "=graph" to indicate that the graph and the relation coincide.

  Before overwriting an existing graph, the system asks the user for confirmation. The relation is displayed in the relation editor window; see Section 3.4.

- PRINT:

  Outputs the graph as an encapsulated postscript-file. The system uses as filename the name of the graph extended by ".gr.eps". The file is created in the directory where the system has been started from.

- MARK NODES:

  Opens a pop up window which allows to enter a relational term. If the value of the term evaluates to a vector with the same number of rows as the number of nodes in the graph, then all nodes belonging to "true-entries" in the vector are "marked", i.e., are drawn as in the above picture indicated. This facility can be used to illustrate computation results, especially subsets of the nodes of the graph. See Section 3.15 for a description of relational terms.

- MARK EDGES:

  Like in the previous case, a pop up window is opened for entering a relational term. If the term evaluates to a homogeneous relation with the same row- and column-number like the number of nodes in the graph and this relation is included in the graph's relation, then all edges in the graph corresponding to "true-entries" in the calculated relation are "marked", i.e., are drawn like shown in the above picture. This facility can be used to illustrate computation results, especially emphasizing subsets of all edges of the graph.

- UNMARK GRAPH:

  Removes all markings of nodes and edges which were added using the two previous menu entries.

- FIT IN WINDOW:

  Adjusts the size of the graph to the size of the window of the graph editor, so that the image of the graph completely fits onto the canvas.

- TOGGLE GRID:

  Switches on or off a grid in the graph editor window. If the grid is displayed, all positions of subsequently placed nodes are aligned to grid positions, i.e., are snapped. Grid positions are the crossings of the grid lines.

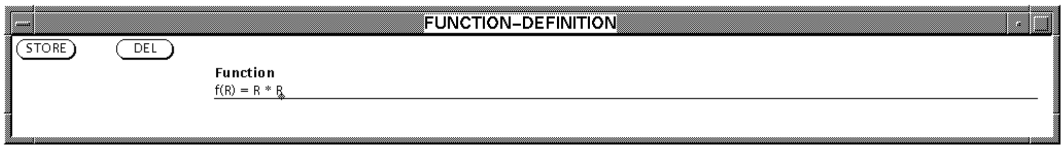
- GRAPH–DRAWING:

  Opens a submenu from which different graph drawing algorithms and a special reflection operation can be choosen. At the time of writing this manual the submenu has the following entries; in the future some other algorithms should be added.

  - SWAP UPPER LOWER: Reflects the graph, i.e., swaps the vertical direction.
  - SPRING (fast): A fast spring embedder.
  - SPRING (slow): A slower spring embedder.
  - LAYER: A layer algorithm which tries to place the edges vertically. It adds special nodes to the graph which are not drawn but allow edges with bends.
  - CORRESPONDENCE: Each node of the graph is drawn twice, i.e., the node set is doubled. Edges are drawn from one node set to the other only.
  - FOREST: This algorithm is only accessible if the graph is a directed forest. The roots of the single directed trees are placed on top of the window.

3.6 The Function-Definition Window

Pressing the button "DEFINE" in the fourth row of the menu window opens a dialog window for defining, editing, and deleting global functions:



Several actions can be invoked:

1. Entering a definition of a function in the input field and pressing the button "STORE" adds this new function to resp. modifies an existing function in the workspace. The syntax of function definitions is described in Section 3.16.

2. A function can be deleted by entering its name in the input field and pressing the button "DEL". The systems asks the user for confirmation before deleting the function definition from the workspace.

3. Selecting a global function from the workspace by clicking onto a list entry in the second scroll list of the directory window, i.e., the list of all global functions, copies the definition of that function into the input field. The function definition can be edited and restored or deleted.

It is important to note that the system only checks the syntax of the entered function but does not validate whether the identifiers contained in the definition denote existing relations, functions, or programs in the workspace. These tests are performed at evaluation time only. Here we want to remark that the decision for implementing this particular behaviour is motivated by the interactive nature of RELVIEW which allows to add objects to or delete them from the workspace at every time.

> **Please note: Function names consist of an identifier plus the left parenthesis. As a consequence, between the proper name of the function, i.e., the identifier, and the opening parenthesis there must be no whitespace.**

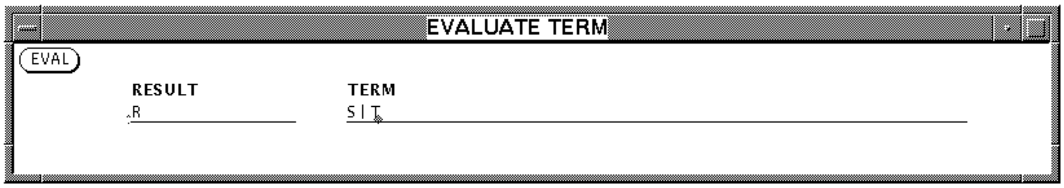
This property takes effect on deleting a function from the workspace. If a global function should be erased from the workspace, a name of the form

$$\gg\text{identifier}\ll\texttt{(}$$

has to be given to the system.

3.7 The Evaluation Window

By pressing the button "EVAL" in the menu window, the term-evaluation window can be opened. Typically it looks as follows:

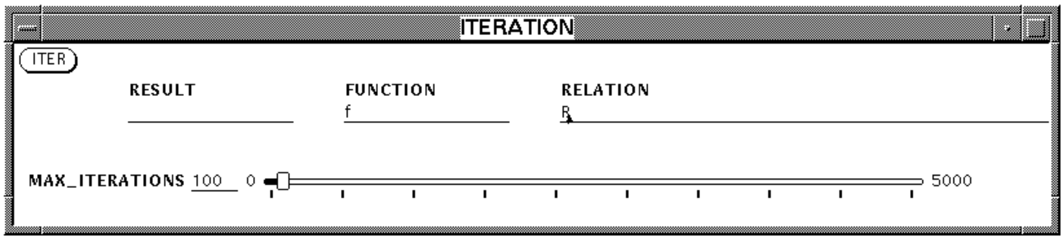


Entering a relational term into the input field "TERM" and a name for the result relation into the field "RESULT" and then pressing the button "EVAL" invokes the evaluation of the given term. Alternatively, the "RETURN"-key on the keyboard starts the computation. This key and the "TAB"-key can be used to switch from the first input field "RESULT" to the second one "TERM". The result, a relation, is stored into the workspace with the entered name. If no result name is given, the default name "$" is used.

In the case of an error, the evaluation stops, a notice prompt is popped up, and an error message is written to the standard output.

3.8 The Iteration Window

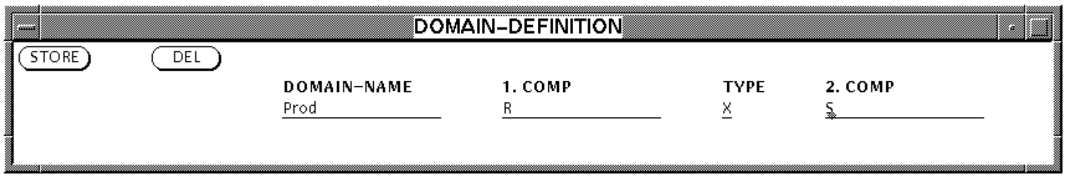
The button "ITER" of the menu window allows to pop up the iteration window which looks as follows:



This window allows the iterated application of a function `f` to a relation `R`, i.e., it computes the sequence `R`, `f(R)`, `f(f(R))`, ... This process stops if the sequence becomes stationary or the maximum number of iteration steps is reached. As the names of the input fields indicate, the function name and the relation name have to be entered into the fields "FUNCTION" resp. "RELATION". In the field "RESULT" a name for the result relation can be entered. If it is omitted, the default name "$" is used. The limit for the number of iteration steps can be set with the slider. The iteration can be invoked by pressing the button "ITER" or the "RETURN"-key on the keyboard. As in the evaluation window, the "RETURN" or the "TAB"-key can be used to switch over from one to the next input field.

3.9 The Domain-Definition Window

Pressing the button "DEF" in the "Domains"-part of the menu window opens the following dialog window which allows to enter a domain definition into the system:



Several actions can be invoked:

1. After entering a name for a relational domain into the field "DOMAIN-NAME" and entering two relational terms describing two relations $R : X \leftrightarrow X$ and $Q : Y \leftrightarrow Y$ on sets X and Y into the input fields "1. COMP" and "2. COMP", respectively, and after choosing the domain type by filling in a "+" (direct sum) or "X"-sign (direct product) into the "TYPE"-field, a press on the button "STORE" puts the corresponding domain definition into the workspace. The defined domain is either the direct sum (ι, κ) with the natural injections $\iota : X \to X + Y$ and $\kappa : Y \to X + Y$ or the direct product (π, ρ) with the natural projections $\pi : X \times Y \to X$ and $\rho : X \times Y \to Y$.

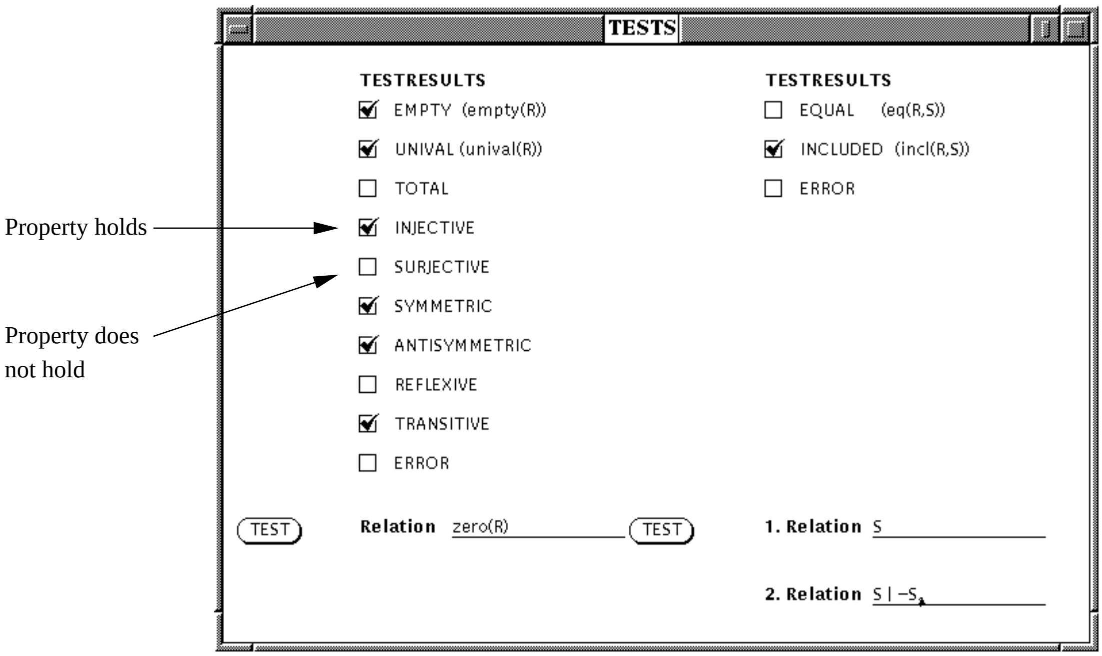
2. Selecting a list entry in the "Domains"-list of the directory window copies the corresponding domain definition into the input-fields of the "DOMAIN-DEFINITION" for editing and restoring or deleting. The domain can be deleted by pressing the button "DEL".

3. Entering a name of a domain contained in the workspace into the "DOMAIN-NAME"-field and pressing the button "DEL" deletes the domain from the workspace.

We note that store actions can also be invoked by pressing the "RETURN"-key on the keyboard. This key and the "TAB"-key allow to switch over to the respective next input field. Before deleting a domain definition from the workspace, the system asks the user for confirmation. As usual, the validity of a domain definiton – the two relational terms describing the different components of the domain have to be denote homogeneous relations – is checked at evaluation time only.

3.10 The Tests Window

Pressing the button "TESTS" in the menu window opens an equally named window which allows performing various kinds of tests on a relation or two relations:



Entering a relational term in the input field "Relation" and pressing the left button "TEST" carries out the following tests on the relation R described by the term:

Test	Relational formula
EMPTY	$R = \mathsf{O}$
UNIVAL	$R^{\mathsf{T}}\, R \subseteq \mathsf{I}$
TOTAL	$R\, \mathsf{L} = \mathsf{L}$
INJECTIVE	$R\, R^{\mathsf{T}} \subseteq \mathsf{I}$
SURJECTIVE	$\mathsf{L} = \mathsf{L}\, R$
ANTISYMMETRIC	$R \cap R^{\mathsf{T}} \subseteq \mathsf{I}$
REFLEXIVE	$\mathsf{I} \subseteq R$
TRANSITIVE	$R\, R \subseteq R$

Filling in two relational terms describing to relations R_1 and R_2 into the input fields "1. Relation" and "2. Relation" and pressing the right button "TEST" performs an equality and an inclusion test on R_1 and R_2, i.e., evaluates $R_1 = R_2$ and $R_1 \subseteq R_2$, respectively.

The results of the tests are indicated by drawing hooks into the boxes corresponding to the different tests. Printing a hook means that the specific property holds. An empty box indicates failure of the corresponding test. If an entered relational term cannot be evaluated, an error is indicated by drawing a hook into the corresponding "ERROR"-box. Switching between the three input fields can be done by pressing the "RETURN"-key on the keyboard. For a more detailed description of the different testable properties see Section 2.2.

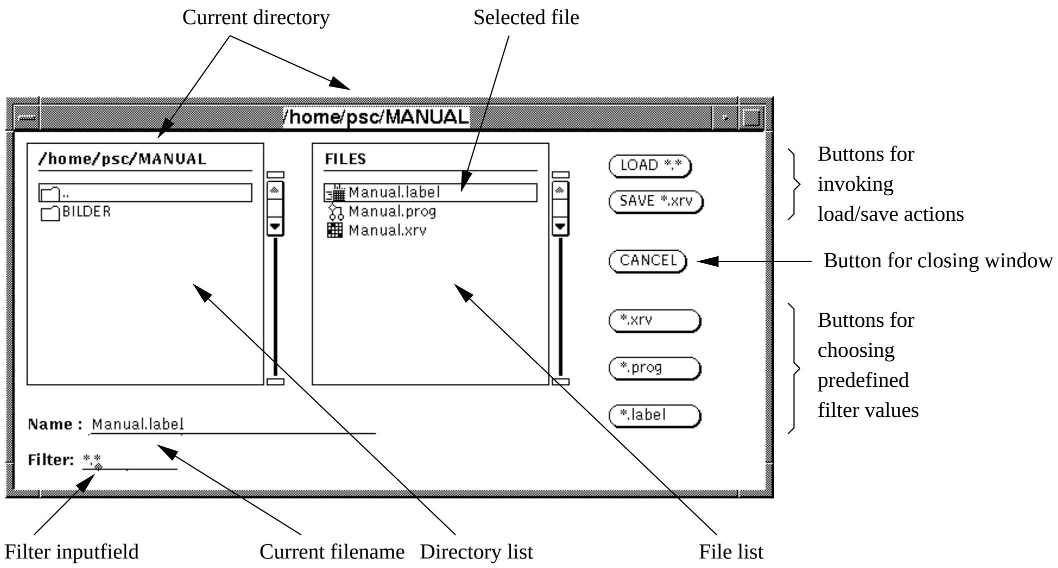
3.11 The File-Chooser

The file-chooser is used to load various kinds of files into the system:

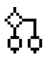
- Xrv-files containing relations, graphs, global functions and domain definitions.
- Program files with the filename extension ".prog" which can hold global function definitions, domain definitions and relational programs.
- Label files with name extension ".label" containing label sets.

In addition, the file-chooser is used to create xrv-files on the disk. The format of program files is described in Section 3.17. A description of the labeling mechanism can be found in Section 3.18.

After pressing the button "FILES" in the menu window, the window of the file-chooser is popped up. The exact appeareance depends on various resources stored in the configuration file, see Section 3.19.1. Typically the window looks as follows:



The title line of the window and the headline of the left scroll list, the directory list for short, show the name of the current directory. The directories contained in the current directory are listed in that scroll list. The list of files which are stored in the actual directory can be found in the right scroll list, the so called file list. Here in general the value of the filter as shown in the "Filter:" input field is considered selecting only a subset of all files contained in the current directory. Depending on the file types, different pictograms are used:

Pictogram	File type
	xrv-file with relations, graphs, functions, and domains
	program file with functions, domains, and programs
	label file containing label sets
	unknown, i.e., all other files

A double click onto a list entry in the directory list selects the corresponding directory as the new current directory. The headlines and the file list are updated. Alternatively, a new current directory can be choosen by entering its name into the input field "Name:" and pressing the "RETURN"-key on the keyboard or the button "LOAD *.*" in the window.

Clicking onto a list entry in the file list copies the name of the choosen file into the input field "Name:". As in case of the directories, a file name can be entered into that input field directly.

Load and save commands on files are invoked by clicking onto the buttons "LOAD *.*" and "SAVE *.xrv", respectively. As shortcuts for the load command the "RETURN"-key on the keyboard can be pressed or a double click onto a file entry in the file list can be performed. The selected file operation always uses the filename which is displayed in the input field "Name:". Please note, that only xrv-files can be saved on disk. If a file is saved, the filename is automatically extended by ".xrv", if the filename extension is not equal to ".xrv".

The button "CANCEL" allows to pop down the file-chooser.

The buttons "*.xrv", "*.prog", and "*.label" chooses predefined filter values. A filter selects a subset of files contained in the actual directory. As filter values regular expressions as used on shell level containing wildcards "?" and "*" are admissible.

3.12 Identifiers and Keywords

A RELVIEW identifier can consist of up to 16 characters. A character can be

- a letter `a`, ..., `z`, `A`, ..., `Z` or
- a digit `0`, ..., `9` or
- an underscore "_".

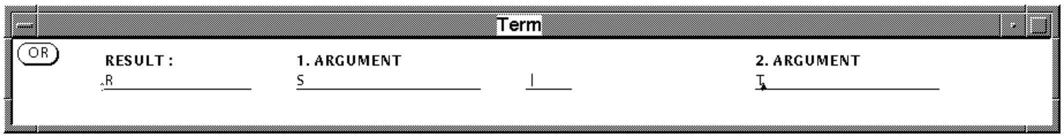
The first character of an identifier has to be a letter as described above.

Not all sequences of characters form legal identifiers. **Names of base functions of RELVIEW are not allowed**. The list of all base functions can be found in the next section. Additionally, some words, in the following called "keywords", are reserved by the system. Keywords of RELVIEW are:

`BEG`, `DECL`, `DO`, `ELSE`, `END`, `FI`, `IF`, `OD`, `PROD`, `RETURN`, `SUM`, `THEN`, `WHILE`

3.13 Base Functions

Some identifiers which are not keywords as listed in the previous section and some special symbols denote predefined functions, the so called base functions of RELVIEW. A subset of all base functions can be accessed by pressing particular buttons in the menu window; see Section 3.2. For example, the window for applying the base function "|" which computes the union of two relations looks as follows (compare with Section 3.7):



The base functions can be divided into several parts:

1. Base functions for calculating constant relations (Section 2.1) and domains:

Syntax	Meaning
L(R)	Universal relation of the same size dimension than R
O(R)	Empty relation of the same size dimension than R
I(R)	Identity relation of the same size dimension than R
Ln1(R)	Universal column vector of the same row number than R
On1(R)	Empty column vector of the same row number than R
L1n(R)	Universal row vector of the same column number than R
O1n(R)	Empty row vector of the same column number than R
dom(R)	Domain R*Ln1(R^) of relation R as column vector

2. Boolean operations (refer to Section 2.1):

Syntax	Button	Meaning
-R	-	Negation (complement) of relation R
R|S	|	Union (join) of R and S
R&S	&	Intersection (meet) of R and S

3. Relationalgebraic operations (see Section 2.1):

Syntax	Button	Meaning
R^	^	Transposition of relation R
R*S	*	Composition (product) of R and S

4. Residuals and symmetric quotients (refer to Section 2.3.2):

Syntax	Button	Meaning
S/R	S/R	Left residual of R and S.
R\S	R\S	Right residual of R and S.
syq(R,S)	SYQ	Symmetric quotient of R and S.

5. Closures (see Section 2.3.1):

Syntax	Button	Meaning
trans(R)	TRANS	Transitive closure of R
refl(R)	REFL	Reflexive closure of R
symm(R)	SYMM	Symmetric closure of R

6. Various operations concerning vectors and points without choice operations (Sections 2.2.3 and 2.4.3):

Syntax	Button	Meaning
inj(v)	INJ	Injection induced by the non-empty vector v
epsi(v)	EPSI	Powerset relation with row number given by the row number of the vector v
init(v)		Initial point of the same dimension than the vector v
succ(v)		Homogeneous successor relation with a dimension given by the number of rows of the vector v
next(p)		Successor of the point p with the same dimension than p

7. Choice operations (refer to Section 2.3.3):

Syntax	Meaning
`point(v)`	A point contained in the non-empty column vector `v`
`atom(R)`	An atom (a pair) contained in the non-empty relation `R`

8. Relational tests on relations (Section 2.3.5):

Syntax	Meaning
`empty(R)`	Test, whether `R` is empty
`unival(R)`	Test, whether `R` is univalent
`eq(R,S)`	Test, whether `R` and `S` are equal
`incl(R,S)`	Test, whether `R` is contained in `S`

9. Functions concerning relational domains (Section 2.4). Most of these base functions take a domain definition as argument, the result however is always a relation.

Syntax	Button	Meaning
`1-st(DD)`	1st	1st component (`DD` domain)
`2-nd(DD)`	2nd	2nd component (`DD` domain)
`p-1(PP)`	P-1	Projection on the 1st component (`PP` product domain)
`p-2(PP)`	P-2	Projection on the 2nd component (`PP` product domain)
`p-ord(PP)`	ORD	Product order (`PP` product domain)
`[R,S]`	TUP	Tupling of relations
`i-1(SS)`	I-1	Injection into 1st component (`SS` sum domain)
`i-2(SS)`	I-2	Injection into 2nd component (`SS` sum domain)
`s-ord(SS)`	ORD	Sum order (`SS` sum domain)
`R+S`	SUM	Sum of relations

10. Base functions concerning function domains (see Section 2.4.4):

Syntax	Button	Meaning
`part-f(R,S)`	PARTF	Columnwise representation of partial functions.
`tot-f(R,S)`	TOTF	Columnwise representation of total functions.

3.14 Operator Precedence and Associativity

The precedence of the unary operators "–" and "^" and the binary infix operators of RELVIEW is as follows (from highest to lowest priority):

<table>
<tr><th>Priority</th><th>Operators</th></tr>
<tr><td>1</td><td>–, ^</td></tr>
<tr><td>2</td><td>*, +</td></tr>
<tr><td>3</td><td>|, &</td></tr>
<tr><td>4</td><td>/, \</td></tr>
</table>

All binary operators and the transposition "^" are left associative. The negation "–" is right associative. Note, that the evaluation order can be changed by using parenthesis. Since evaluation of expressions is done from left to right, in the case of equal priorities of operators sometimes even parenthesis have be used.

3.15 Relational Terms

The main purpose of RELVIEW is the evaluation of relational terms, also called relational expressions. A relational term can be of one of the following syntactical forms.

1. It can be a RELVIEW identifier, i.e., of the shape $\gg$ ident $\ll$.

   The identifier can denote a relation or a parameter of a global function defined in the workspace. In the context of a relational program additionally it can be a formal parameter or a local variable.

2. It can be an application of a base function, i.e., it looks as follows:

   (a) `-`$\gg$ term $\ll$
   (b) $\gg$ term $\ll$`^`
   (c) $\gg$ term $\ll$ $\gg$ infix base operation $\ll$ $\gg$ term $\ll$
   (d) $\gg$ relational base function $\ll$ ($\gg$ term $\ll$, ..., $\gg$ term $\ll$)
   (e) $\gg$ domain base function $\ll$ ($\gg$ domain name $\ll$)

   Here a relational base function is a base function of RELVIEW taking relational arguments. In contrast, a domain base function takes a name of a relational domain defined in the workspace. The base functions of RELVIEW are described in Section 3.13.

3. It can be an application of a global function defined in the workspace or a local function declared in a relational program, i.e., it is of the following form:

   $\gg$ function name $\ll$ ($\gg$ term $\ll$, $\gg$ term $\ll$, ... , $\gg$ term $\ll$)

   We note that a function name simply is a RELVIEW identifier as described in Section 3.12.

4. It can be a call of a relational program, i.e., it looks as follows:

   $\gg$ program name $\ll$ ($\gg$ term $\ll$, $\gg$ term $\ll$, ... , $\gg$ term $\ll$)

   As in the case of an application of a user-defined function, a program name is a RELVIEW identifier.

As usual, parenthesis "(", ")" can be used to force an evaluation order. Examples for relational terms are:

```
R   -Q   Relation_45|X   f(R, S&g(xyz_1))   p-1(Prod)   clear(R^)
```

The evaluation of a relational term is based on a Call-by-value strategy. Please note that the evaluation of relational terms strongly depends on the contents of the workspace. For example, if an identifier contained in a term denotes a relation, a global function or a program which cannot be found in the workspace at evaluation time, the computation will fail. The same is true for applications of base functions which take a domain name as an argument. Here additionally the domain definition has to be valid. In all these error cases, the system presents a notice prompt and writes an error message to standard output.

3.16 Relational Functions

A definition of a relational function is of the following syntactic form:

$$\gg \text{function name} \ll (\gg \text{ident} \ll, \gg \text{ident} \ll, \ldots, \gg \text{ident} \ll) \;\texttt{=}\; \gg \text{term} \ll$$

The function name and the formal parameters inside the brackets can be RELVIEW identifiers as described in Section 3.12.

Please note: The left parenthesis belongs to the function name. As a consequence between the proper name and the left parenthesis there must be no whitespace.

The relational term on the right hand side can not only contain global relations, function applications, and program calls, but formal parameters occuring in the parameter list as well. Examples for relational functions are:

```
f(X)=X|-X   mult_R(X)=X*R   h23(X1,Q)=(clear(Q)/R)&trans(X1)
```

Relational functions can be entered into the system using the "FUNCTION-DEFINITION"-window. Alternatively, a function definition can be written into a program file and loaded into the system using the file-chooser, see Section 3.17.3. In the last case, the definition has to be terminated by a dot ".".

3.17 Relational Programs

A relational program in the sense of the RELVIEW system essentially is a while-program based on the datatype of binary relations. Such a program has many similarities with a function procedure in the programming languages PASCAL or MODULA-2. The execution is based on a Call-by-value, i.e., leftmost-innermost, strategy. The system uses static binding and the usual scoping rules. Instead of presenting the RELVIEW programming language in a strong formal manner, we follow a more pragmatical point of view by introducing the syntax of the different program constructs in an intuitive way. At this point we want to remark that examples of relational programs can be found in Section 4 below. The syntactic form of a relational program can be sketched as follows:

```
≫ident≪(≫ident≪, ≫ident≪, ..., ≫ident≪)
 DECL ≫declaration of local domains≪
      ≫declaration of local functions≪
      ≫declaration of local variables≪
 BEG  ≫statement≪ ;
         ...
      ≫statement≪
      RETURN ≫term≪
 END.
```

As it is shown, a relational program starts with a head line containing the program's name and a list of formal parameters. After the keyword DECL then the declaration part follows. It consists of the declarations of relational domains, followed by the declarations of the local functions, and, finally, of the local variables for relations. The third part of a relational program is its body, the keyword `BEG` followed by a sequence of statements which are separated by semicolons. Relational programs compute values. Hence, the last part of such a program is the `RETURN`-clause, which consists of the keyword `RETURN` followed by a relational term. The keyword `END` and a dot behind `END` indicate the end of a relational program.

The shape of admissible identifiers, denoted by $\gg$ident$\ll$ in the above picture, is described in Section 3.12.

3.17.1 Structure of the Declaration Part

The single declarations of local domains and local functions are terminated by semicolons. Local variables are separated by commas.

- Domain declarations introduce binary relational products and binary relational sums together with some natural relations like projections and injections as described in Section 2.4. A declaration of a binary relational direct product is of the following form:

  $\gg$ident$\ll$ `= PROD(`$\gg$term$\ll$`,`$\gg$term$\ll$`) ;`

  A binary relational sum is declared as follows:

  $\gg$ident$\ll$ `= SUM(`$\gg$term$\ll$`,`$\gg$term$\ll$`) ;`

  In the arguments of the domain formers `PROD` and `SUM` also parameters of the superior relational program may occur. As in case of domain definitions entered into the workspace using the "DOMAIN-DEFINITION"-window, the admissibility of the arguments – both have to represent homogenous relations – are checked at evaluation resp. execution time only.

- A declaration of a local relational function is of a similar form as a definition of a global function entered into the system with the help of the "FUNCTION-DEFINITION"-window. See Sections 3.6 and 3.16. However, each declaration has to be terminated by a semicolon:

  $\gg$ident$\ll$`(`$\gg$ident$\ll$`, ... ,`$\gg$ident$\ll$`) =` $\gg$term$\ll$ `;`

  The declaration consists of a function name, a list of formal parameters, and a relational term. Please note that the term may not only refer to objects stored in the workspace but may also contain parameters of the superior relational program and locally declared domains and functions. But on no account does an identifier which occurs in the relational term refer to a local variable of a program. As in case of the local domain declarations, it is only checked at execution time and not at load time, whether a call of a local declared function can be evaluated, i.e., whether all functions and relations contained in the term are defined.

- Declarations of local variables look as follows:

$$\gg \text{variable name } 1 \ll \texttt{,}$$
$$\gg \text{variable name } 2 \ll \texttt{,}$$
$$\ldots$$
$$\gg \text{variable name n} \ll$$

As sketched above, the variable names are separated by a comma. Each variable name can be a RELVIEW identifier as described in Section 3.12. **Local variables are not implicitly initialized.** Before using, i.e., reading, a variable, a value has to be assigned to it. Assignment statements are described in the next section.

3.17.2 Syntax of Statements

Essentially, the body of a relational program is a sequence of statements as sketched above at the beginning of this section. In the following we describe the syntactic structure of the statements of the RELVIEW programming language.

1. The simple statements of the language are the *assignments* of the form

   $$\gg \text{ident} \ll \texttt{=} \gg \text{term} \ll$$

   with a **local variable on the left-hand side** and a relational term on the right-hand side. Note that the term may contain calls of relational programs. In particular, recursion is allowed. Since on the left-hand side of an assignment only local variables are allowed, the execution of a program produces no side-effects with respect to relations stored in the workspace. As usual, it is checked at execution time only whether all pieces contained in the term are available in the workspace resp. locally declared.

2. *Sequential composition* is the first kind of compound statements. It is denoted by a semicolon:

   $$\gg \text{statement} \ll \texttt{;} \gg \text{statement} \ll$$

3. Like in PASCAL and MODULA-2, there are two different kinds of *conditional statements* in the programming language of RELVIEW. Without an else part a conditional looks like this:

   $$\texttt{IF} \gg \text{term} \ll \texttt{THEN} \gg \text{statement} \ll \texttt{FI}$$

   An else part in a conditional is also allowed. This two-sided conditional then looks as follows:

   $$\begin{aligned}\texttt{IF} \gg \text{term} \ll \texttt{THEN} &\gg \text{statement} \ll \\ \texttt{ELSE} &\gg \text{statement} \ll \texttt{FI}\end{aligned}$$

   At execution time the value of the relational term in a conditional statement has to be a relation of type $[\mathbf{1} \leftrightarrow \mathbf{1}]$, in RELVIEW represented by a 1×1-matrix. Compare Section 2.3.5.

4. The *while-loop* of the programming language of the RELVIEW system has the following form:

   $$\texttt{WHILE} \gg \text{term} \ll \texttt{DO} \\ \gg \text{statement} \ll \texttt{OD}$$

   As in the case of conditional statements at execution time the value of the condition of the loop has to be a relation of type $[\mathbf{1} \leftrightarrow \mathbf{1}]$, i.e., a 1×1-matrix.

   Please note: "`RETURN` $\gg$ term $\ll$" is not a statement. Furthermore, note that a skip statement is not part of the RELVIEW programming language. As a consequence, before the key words `ELSE`, `FI`, `OD` and the `RETURN`-clause there must not be a semicolon.

3.17.3 Program Files

Relational programs are stored in so called program files, or prog-files for short. Program files are human-readable text files which can be created using a text-editor. They are loaded into the system with the help of the file-chooser, see Section 3.11. Relational programs are written into a program file using a format as sketched above at the beginning of Section 3.17. Beside relational programs a program file can contain declarations of global functions and global domain definitions as well. The format for these two types of declarations are similar to the local declarations which can occur in the declaration part of a relational program as described in Section 3.17.1. However, each declaration has to be terminated by a dot ".":

Type of declaration	Syntactic form
Direct product	$\gg$ ident $\ll$`=PROD(`$\gg$ term $\ll$`,`$\gg$ term $\ll$`) .`
Direct sum	$\gg$ ident $\ll$`=SUM(`$\gg$ term $\ll$`,`$\gg$ term $\ll$`) .`
Global function	$\gg$ ident $\ll$`(`$\gg$ ident $\ll$`, . . . ,`$\gg$ ident $\ll$`)=`$\gg$ term $\ll$`.`

Furthermore, comments can be added everywhere in a program file. Like in PASCAL and Modula-2, a comment must be enclosed by the brackets "`{`" and "`}`". Encapsulation of comments within a comment is not allowed.

While loading a program file, the RELVIEW system performs various syntax checks. If a syntax error is detected, the program file is rejected, the error condition is indicated by a notice prompt and an error message is written to standard output. The system prints out the line number, or at least an estimation, where the error occured.

Please note: Program files can be loaded into the system, but cannot be written back to disk.

At the end of this section we present a small example of a program file containing declarations of a relational domain, a global function and, finally, a relational program for the computation of the transitive closure R^+ of a relation R by calculating the least fixpoint μ_f of the function

$$f : [X \leftrightarrow X] \to [X \leftrightarrow X] \qquad\qquad f(Q) = R \cup Q\,R\,.$$

For illustration purposes, besides two variables in the program `TransClosurs` also a local declared function `mult` is used and some comments are added:

```
Prod = PROD(R,S).              { domain declaration }

zero(R) = R&-R.                { global function }

TransClosure(R)                { relational program }
  DECL mult_R(X) = X * R;      { local function }
       res, X                  { local variables }
  BEG  X = R;
       res = zero(R);
       WHILE -incl(X,res) DO
          res = res | X;
          X = mult_R(X) OD     { X = X * R }
       RETURN res
 END.
```

More examples for relational programs can be found in Section 4.

3.18 Labels

The RELVIEW system provides a mechanism for labeling rows and columns of relations and nodes of graphs. A label is simply a RELVIEW identifier. Labels are only used for illustration purposes. Adding labels to rows and/or columns of relations and nodes of graphs often increases the readability and understandability of relations and graphs. **Labels do not carry a semantics within the system**. Especially the evaluation of relational terms does not depend on values of labels.

> **Please Note: In no cases are labels attached to relations or graphs or informations about the labeling of a relation or a graph written to a xrv-file.**

3.18.1 Label Sets and Label Files

Labels are organized in so called "label sets". A label set is simply a named mapping from natural numbers to labels, i.e., identifiers. A definition of a label set is of the following form:

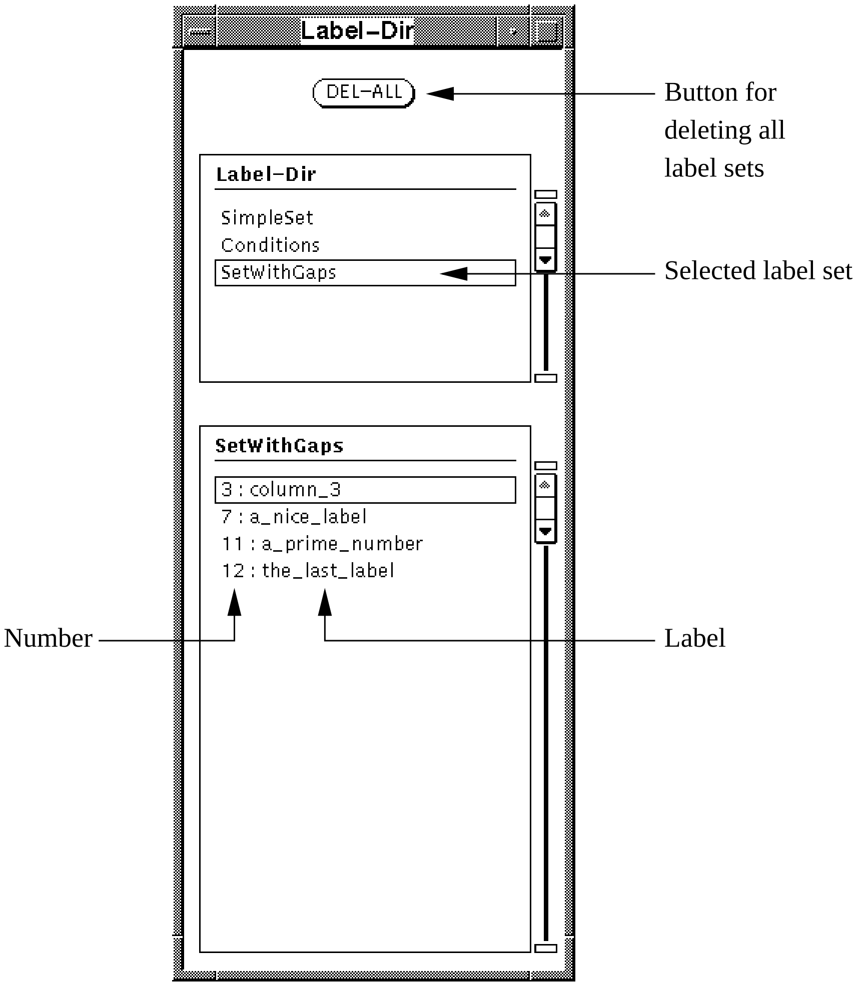
$$\begin{array}{lll} \gg\text{label-set-name}\ll \; \texttt{=} & \{ & \gg\text{number}\ll \; \gg\text{label}\ll, \\ & & \gg\text{number}\ll \; \gg\text{label}\ll, \\ & & \vdots \\ & & \gg\text{number}\ll \; \gg\text{label}\ll \\ & & \gg\text{number}\ll \; \gg\text{label}\ll \\ & \} & \end{array}$$

The name of a label set is an ordinary RELVIEW identifier. Please note, that the blank between a number and a label acts as a separator. The numbers in a label set definition have to be unique but are allowed to appear in any order. In addition, the numbers in a label set do not have to build up an intervall. In fact, any gaps in the numbering are allowed. Some examples of label sets can be found in Section 3.18.6.

Label sets are stored in ordinary text files with the filename extension ".label", which are called label files for short. One label file can contain an arbitrary number of definitions of label sets. A label file, more exactly spoken the label sets stored in the file, can be loaded into the RELVIEW system by using the file-chooser, see Section 3.11.

3.18.2 The Label Directory Window

The label sets loaded into the system can be inspected by the help of the label directory window. This window can be opened by pressing the button "LABEL" in the third button row of the menu window. Typically this window looks as follows:



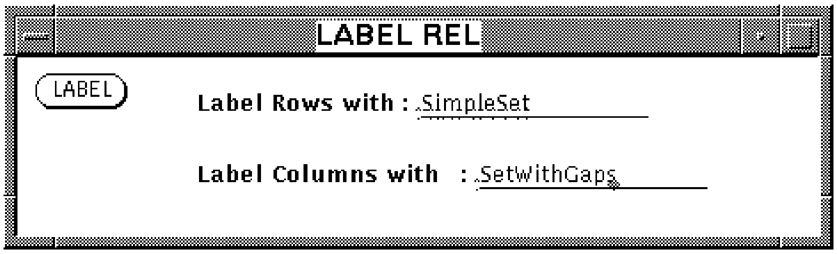
As the picture shows, the label directory window contains two scroll lists. In the first list, the names of the loaded label sets are shown. A label set can be selected by clicking onto the corresponding list entry.

The second scroll list displays the selected label set. The set is printed as a list of items, sorted by numbers, of the form "Number : Label".

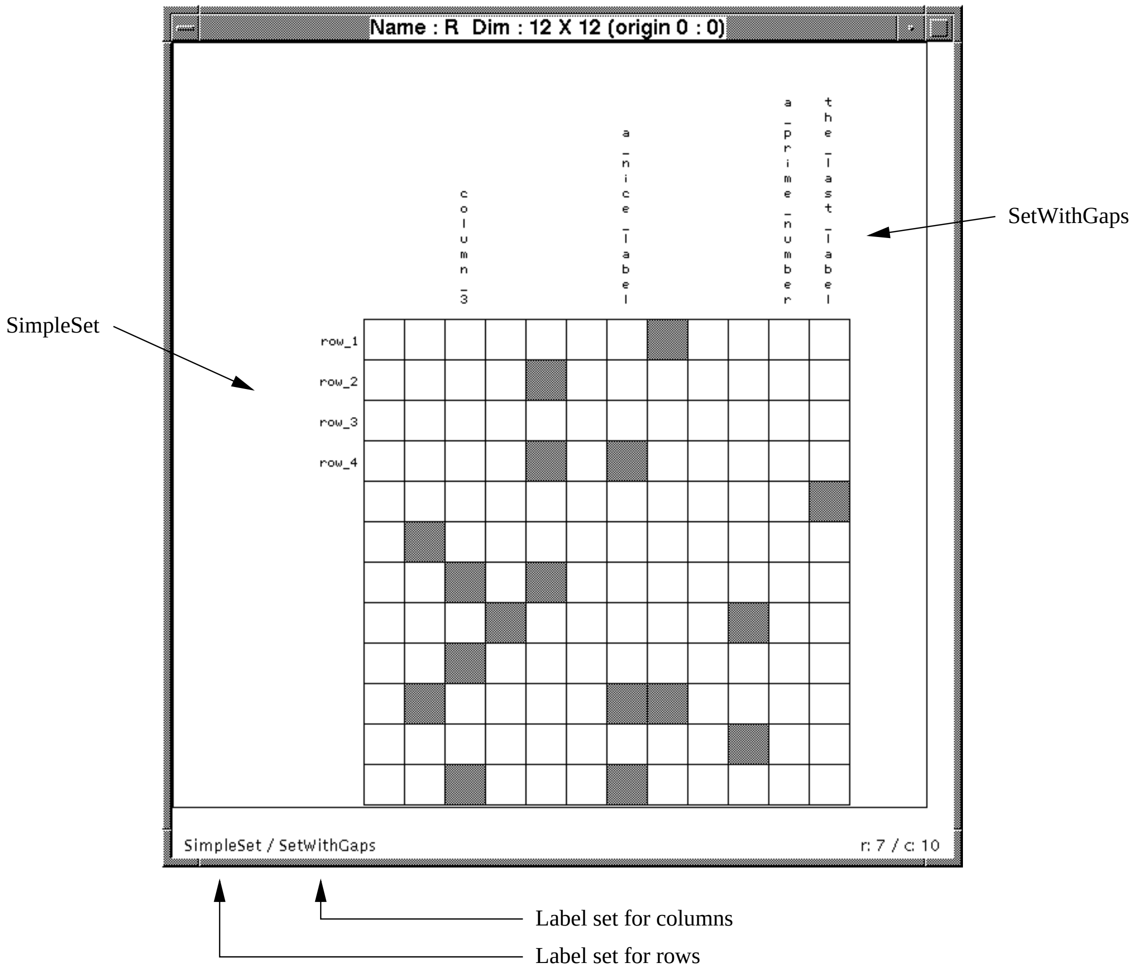
3.18.3 Attaching Labels to Relations

The rows and/or the columns of a relation can be labeled with different label sets. You can select the desired label sets by choosing the menu item "LABEL REL" in one of the submenus of the relation editor, see Section 3.4.2.

Choosing the sketched menu item opens a pop up window with two input fields for entering one or two names of label sets for the rows and/or the columns of the relation:



After pressing the push button "LABEL" in this pop up, the rows and/or the columns of the relation in the relation editor are attached with the labels of the selected label sets. For example, the window of the relation editor showing a labeled relation can look as follows:



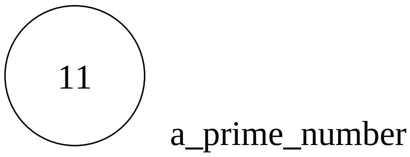
Please note: The number of labels in a label set which should be used for labeling the rows or columns of a relation must not exceed the number of rows resp. columns of that relation.

3.18.4 Removing Labels from a Relation

The labels attached to the rows and/or columns of a relation can be removed by selecting the menu item "UNLABEL REL" in a submenu of the relation editor. Please refer to Section 3.4.2 for a description of the menu structure of the relation editor.

3.18.5 Labeling Graphs

Graphs, more exactly spoken the nodes of a graph, cannot be labeled directly. Instead you can label the homogeneous relation belonging to the graph. Then the labels added to the relation are attached to the nodes of the graph automatically. A labeled node looks as follows:



Please note: If you want to label the nodes of a graph, the rows and the columns of the corresponding relation have to be labeled with the same label set.

3.18.6 Examples of Label Sets

In this section we show some examples of label sets for illustration purposes:

```
SimpleSet =   { 1 row_1,
                2 row_2,
                3 row_3,
                4 row_4
              }
SetWithGaps = { 3 column_3,
                11 a_prime_number,
                7 a_nice_label,
                12 the_last_label
              }
Conditions =  { 1 Thinking_P1,
                2 Thinking_P2,
                3 Thinking_P3,
                4 Eating_P1,
                5 Eating_P2,
                6 Eating_P3
              }
```

3.19 Miscellaneous

This section deals with the configuration of RELVIEW, explains how to use a startup file and gives some hints about the installation of the system. In particular, the URL of the RELVIEW Web-page can be found in Section 3.19.3.

3.19.1 The Configuration of RELVIEW

The RELVIEW system is in a wide range user-configurable. Fonts, the layout of several windows, button sizes and more can be choosen by defining resources in a resource-file. The name of that file is always ".xrelview". Please note the dot at the beginning of the filename.

At startup time, the system looks for a configuration file at two different places: First, the current directory is scanned. If it does not contain a configuration file, the user's home directory is searched for.

If in both directories no ".xrelview" file can be found by the system, the system uses default values for the various resources. The resources supported by RELVIEW are described in detail in appendix A.

3.19.2 Using a Start-up File

The RELVIEW system allows to load a set of relations, functions and domain definitions automatically at startup time. The loaded objects are declared as "hidden" by the system with the effect that they are only listed in the directory window, if the user selects the menu item "HIDDEN" in the "SHOW NOW"-menu of the directory window. For details see Section 3.3.

At startup time, first the system looks for a file "start_up.xrv" in the current directory. If this file cannot be found, in a second step the user's home directory is searched for a file ".start_up.xrv". Please note the dot at the beginning of the filename of the startup file located in the home directory.

Startup files can easily be created by the user herself. We recommend to write all definitions of desired global functions and relational domains into a prog-file, load this file using the file-chooser into the (naked) system and save it as a xrv-file with name "start_up.xrv" or ".start_up.xrv", respectively.

In appendix B an example for a set of functions which are normally stored in a startup file can be found.

3.19.3 Installation of RELVIEW

The RELVIEW system is freely available by FTP from host

ftp.informatik.uni-kiel.de.

It is located in the following directory:

pub/kiel/relview

Two different ports of the system are available, namely versions for

1. Sun SPARC workstations running Solaris 2.5 and
2. INTEL-based Linux systems (Kernels 1.2.x, 1.3.x, and 2.0.x).

Detailed information about the directory structure and the available files on the FTP-server can be found in a README file contained in the directory pub/kiel/relview. Additional informations and latest news about RELVIEW are published on the World-Wide-Web at the following location:

URL: http://www.informatik.uni-kiel.de/~progsys/relview.html

4 Examples for the Use of RELVIEW

In this section, we show how to specify and develop algorithms for discrete structures in the relation-algebraic framework such that the results can directly be executed in RELVIEW. Firstly, we deal with a lattice-theoretical application, viz. the computation of the cut completion of a partially ordered set. Then, we apply relational algebra for the analysis of Petri nets. Using RELVIEW, here we investigate in particular the dining philosophers condition/event net. And, finally, we show how to solve some graph-theoretic problems using relational algebra and the RELVIEW system.

4.1 A Lattice-Theoretic Application

In classical mathematics, the method of Dedekind cuts in the rational numbers is one of the ways for introducing the real numbers. It has been generalized to a procedure for constructing completions of arbitrary partially ordered sets. Based on [4], in the following we show how such a cut completion can be treated in the calculus of relations and computed using RELVIEW. To obtain this, we have to distinguish between the set-theoretic symbols $\in$ and $\subseteq$ on the meta-level and the membership relation respectively the set inclusion relation on the object-level. In the sequel, we use on the object level the two relations $\varepsilon : X \leftrightarrow 2^X$ and $\sqsubseteq : 2^X \leftrightarrow 2^X$ for membership and inclusion.

4.1.1 Cut Completion of a Partially Ordered Set

Assume $R : X \leftrightarrow X$ to be a partial ordering. We call the pair (X, R) a partially ordered set. If, in addition, every set $Y \in 2^X$ has a least upper bound and a greatest lower bound, then (X, R) is called a *complete lattice*. The method of Dedekind cuts for constructing a completion of the partially ordered set (X, R) is as follows (compare [14, 11]): For a given set $Y \in 2^X$ one considers two sets, viz. $Mi(R, Y)$, the set of all lower bounds of Y wrt. R, and $Ma(R, Y)$, the set of all upper bounds of Y wrt. R. Then one defines a set $C \in 2^X$ to be a *Dedekind cut* if $Mi(R, Ma(R, C)) = C$.

Obviously, for each element $y \in X$ the set $(y) := \{x \in X : R_{xy}\}$ is a Dedekind cut, called the *principal cut* generated by y.

Let $\mathcal{C}$ denote the set of Dedekind cuts of X and $\mathcal{P}$ denote the set of principal cuts of X. Furthermore, let $\sqsubseteq_{\mathcal{C}} : \mathcal{C} \leftrightarrow \mathcal{C}$ and $\sqsubseteq_{\mathcal{P}} : \mathcal{P} \leftrightarrow \mathcal{P}$ denote the restrictions of the set inclusion relation $\sqsubseteq : 2^X \leftrightarrow 2^X$ to the Dedekind cuts and principal cuts, respectively. Then $(\mathcal{C}, \sqsubseteq_{\mathcal{C}})$ is a complete lattice, having $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$ as sub-ordering. Furthermore, the function $x \mapsto (x)$ is an order isomorphism between (X, R) to $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$. Therefore, the lattice $(\mathcal{C}, \sqsubseteq_{\mathcal{C}})$ is said to be the *cut completion* of the partially ordered set (X, R).

4.1.2 A Relation-Algebraic Approach to Cut Completion

For a relation-algebraic construction of the cut completion $(\mathcal{C}, \sqsubseteq_{\mathcal{C}})$ of the partially ordered set (X, R), we start with the definition that an element $y \in X$ is a lower bound of the set $Y \in 2^X$ if and only if $\forall\, z\; z \in Y \rightarrow R_{yz}$. Then, we describe Y by a vector $v : X \leftrightarrow \mathbf{1}$ and use the component-wise notation for the left residual given in Section 2.3.2. We obtain that the set of all lower bounds of v wrt. R is expressed by the vector $mi(R, v) = R/v^{\mathsf{T}}$.

Transposing the relation R yields $ma(R,v) = mi(R^{\mathsf{T}}, v)$ as the vector of all upper bounds of v wrt. R. In the language of RELVIEW, we obtain the relational functions

```
mi(X,Y) = X / Y^.
ma(X,Y) = mi(X^,Y).
```

for mi and ma. If the second argument of `mi` resp. `ma` is not a vector but an arbitrary relation, then obviously the functions compute lower and upper bounds column-wise.

Aiming at the computation of a vector describing all Dedekind cuts, next we consider a set $C \in 2^X$. Using the correspondences between certain kinds of logical and relation-algebraic constructions, we obtain

$$\begin{aligned}
& C \text{ is a cut} \\
\iff & \forall\, x\ x \in Mi(R, Ma(R,C)) \leftrightarrow x \in C \\
\iff & \forall\, x\ mi(R, ma(R,\varepsilon))_{xC} \leftrightarrow \varepsilon_{xC} \\
\iff & \mathsf{syq}(mi(R, ma(R,\varepsilon)), \varepsilon)_{CC} \\
\iff & \exists\, M\ \mathsf{syq}(mi(R, ma(R,\varepsilon)), \varepsilon)_{CM} \wedge C = M \\
\iff & \exists\, M\ \mathsf{syq}(mi(R, ma(R,\varepsilon)), \varepsilon)_{CM} \wedge \mathsf{I}_{CM} \wedge \mathsf{L}_M \\
\iff & ((\mathsf{syq}(mi(R, ma(R,\varepsilon)), \varepsilon) \cap \mathsf{I})\, \mathsf{L})_C\,.
\end{aligned}$$

Here the types of the identity relation respectively the universal relation are $\mathsf{I} : 2^X \leftrightarrow 2^X$ and $\mathsf{L} : 2^X \leftrightarrow \mathbf{1}$. Now, we remove the subscript C in the result of the above derivation and arrive at the relation-algebraic description

$$CutVector(R) = (\mathsf{syq}(mi(R, ma(R,\varepsilon)), \varepsilon) \cap \mathsf{I})\, \mathsf{L} : 2^X \leftrightarrow \mathbf{1}$$

of the vector describing the subset $\mathcal{C}$ of 2^X (in the sense of Section 2.2.3) the members of which are the Dedekind cuts. Using the injective mapping $\mathsf{inj}(CutVector(R)) : \mathcal{C} \leftrightarrow 2^X$ given by this vector (see again Section 2.2.3) in combination with the membership relation $\varepsilon : X \leftrightarrow 2^X$, we obtain the elements of $\mathcal{C}$ as the colums of the relation

$$CutRelation(R) = \varepsilon\ \mathsf{inj}(CutVector(R))^{\mathsf{T}} : X \leftrightarrow \mathcal{C}\,.$$

Based on the above relational functions `mi` and `ma`, in RELVIEW the computation of the vector describing the Dedekind cuts respectively the column-wise representation of the Dedekind cuts look as follows:

```
CutVector(R)
  DECL Id, c, eps
  BEG  eps = epsi(dom(R));
       Id = I(eps^ * eps);
       c = dom(syq(mi(R,ma(R,eps)),eps) & Id)
       RETURN c
  END.

CutRelation(R) = epsi(dom(R)) * inj(CutVector(R))^.
```

Since the Dedekind cuts are ordered by set inclusion, in the third step of our cut completion procedure we consider the inclusion relation $\sqsubseteq : 2^X \leftrightarrow 2^X$. If we use the correspondences between the two relations $\sqsubseteq$ and ε and the meta-level symbols $\subseteq$ and $\in$, then we obtain that set inclusion equals as a relation on the object level the right residual $\varepsilon \setminus \varepsilon$. Hence, the two RELVIEW base functions `epsi` and `\` suffice to compute it. Next, we consider the injective mapping $\mathsf{inj}(\mathit{Cutvector}(R)) : \mathcal{C} \leftrightarrow 2^X$ which represents $\mathcal{C}$ as a subset of 2^X. It is obvious that the restriction of set inclusion to the Dedekind cuts can be described as

$$\sqsubseteq_{\mathcal{C}} = \mathsf{inj}(\mathit{Cutvector}(R)) \sqsubseteq \mathsf{inj}(\mathit{Cutvector}(R))^{\mathsf{T}} : \mathcal{C} \leftrightarrow \mathcal{C}\,.$$

A transformation of this equation into a relational program `CutLattice` (which avoids computing the injective mapping twice) is obvious. We obtain:

```
CutLattice(R)
  DECL emb, eps, incl
  BEG  eps = epsi(dom(R));
       incl = eps \ eps;
       emb = inj(CutVector(R))
       RETURN emb * incl * emb^
  END.
```

As the last step of relational cut completion it remains to describe the injective order homomorphism $x \mapsto (x)$ from X to $\mathcal{C}$ with relation-algebraic means. If we use the common function notation for $\imath := \mathsf{inj}(\mathit{CutVector}(R))$, then we obtain

$$\begin{aligned}
& (x) = \imath(C) \\
\iff & \forall\, y\ y \in (x) \leftrightarrow y \in \imath(C) \\
\iff & \forall\, y\ R_{yx} \leftrightarrow y \in \imath(C) \\
\iff & \forall\, y\ R_{yx} \leftrightarrow (\exists\, M\ y \in M \wedge \imath(C) = M) \\
\iff & \forall\, y\ R_{yx} \leftrightarrow (\varepsilon\, \imath^{\mathsf{T}})_{yC} \\
\iff & \mathsf{syq}(R, \varepsilon\, \imath^{\mathsf{T}})_{xC}\,.
\end{aligned}$$

By a removal of the subscripts x and C, from the result of this derivation we get

$$\mathit{Embedding}(R) = \mathsf{syq}(R, \varepsilon\, \mathsf{inj}(\mathit{CutVector}(R))^{\mathsf{T}}) : X \leftrightarrow \mathcal{C}$$

as relation-algebraic description of the injective mapping which associates an element $x \in X$ to a Dedekind cut $C \in \mathcal{C}$ if and only if C is the principal cut generated by x. In RELVIEW, this mapping is implemented by the relational function

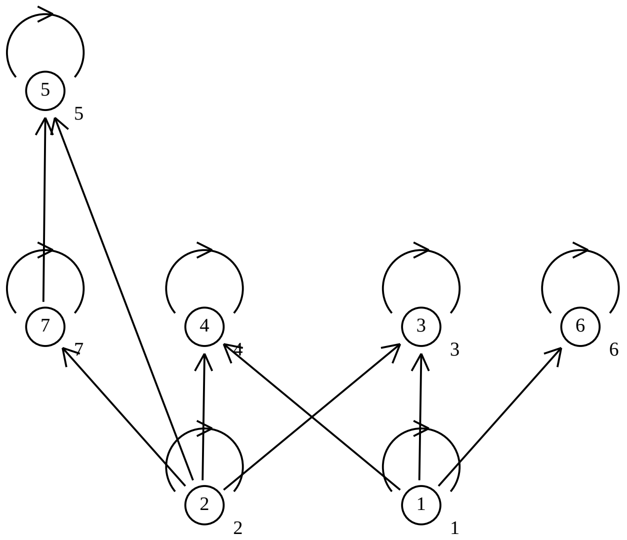
```
Embedding(R) = syq(R,epsi(dom(R)) * inj(CutVector(R))^).
```

using again the above relational program `CutVector`.

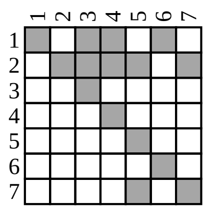
4.1.3 An Example

After having presented a procedure for constructing the cut completion of a partially ordered set with relation-algebraic means and the corresponding RELVIEW functions respective programs `mi` through `Embedding` in the previous section, we now deal with a concrete example.

We consider a set X with 7 elements, for simplicity numbered by 1 through 7, and a partial ordering $R : X \leftrightarrow X$, which, as a directed graph R produced on the window of the graph editor using the layer graph drawing algorithm of RELVIEW, looks as follows:

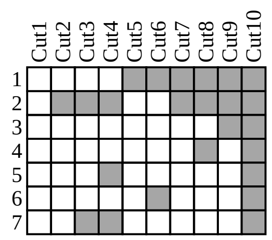


To compute the cut completion using RELVIEW, we have to create a relation from this directed graph with the same name R. The following picture shows this relation as 7×7 Boolean matrix as presented on the window of the relation editor:

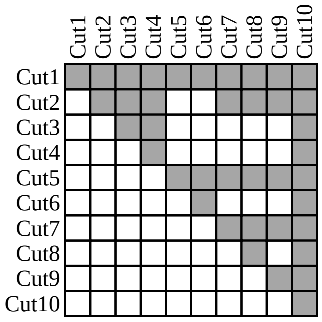


Since the nodes of the above directed graph are labeled, the rows and columns of this Boolean matrix are also labeled.

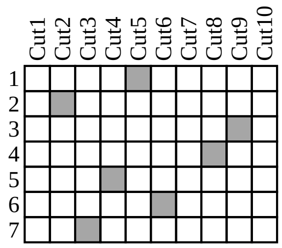
Now, we assume a program file containing the relational functions and programs of the last section and that this file is loaded into the system's workspace. If we evaluate the relational term `CutRelation(R)` and then add labels to the rows as well as the columns of the result, we get the following column-wise representation of the 10 Dedekind cuts:



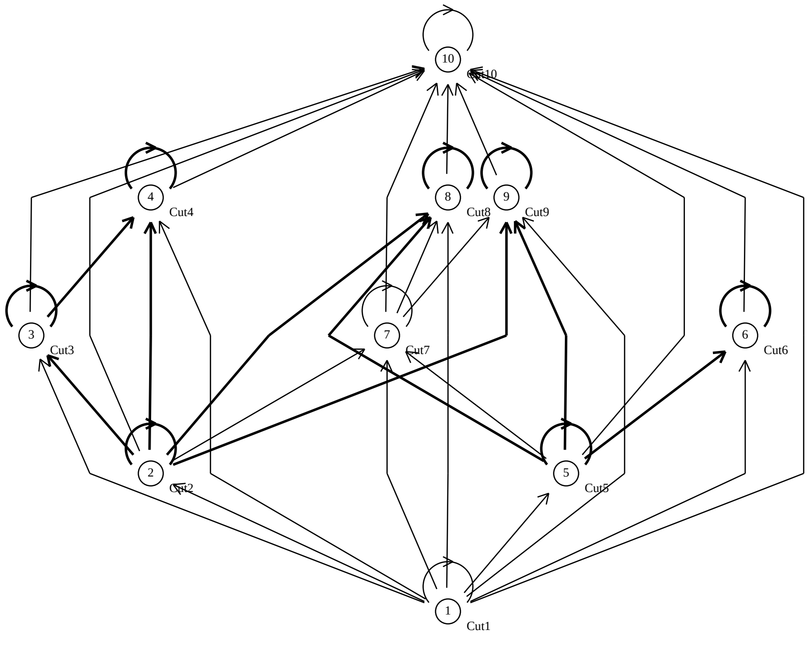
I.e., the set $\mathcal{C}$ of Dedekind cuts consists of $\emptyset$, $\{2\}$, $\{2,7\}$, $\{2,5,7\}$, $\{1\}$, $\{1,6\}$, $\{1,2\}$, $\{1,2,4\}$, $\{1,2,3\}$, and the entire set X. The next picture of this section shows the Boolean matrix representation of the ordering relation of the complete lattice $(\mathcal{C}, \sqsubseteq_{\mathcal{C}})$, which is obtained by evaluating the relational term `CutLattice(R)`. For illustration purposes, again labels are added to the rows and columns of the result.



Finally, we want to visualize this 10×10 Boolean matrix with RELVIEW as a directed graph, where additionally the embedding of the original partially ordered set is indicated by marked (i.e., boldface) edges. To this end, first, we compute the injective mapping which embedds X in $\mathcal{C}$ by evaluating the relational term `Embedding(R)` and store the result in the workspace of RELVIEW with the name `E`. The labeled 7×10 Boolean matrix representation of `E` looks as follows:



Then, we draw the ordering relation of the complete lattice as a directed graph. Finally, we mark the edges of this directed graph with the relational term `E^*R*E`, since the value of this term is the sub-relation of the cut ordering generated by the images of X. The result is shown in the next picture, for the production of which again the layer graph drawing algorithm of RELVIEW has been used.



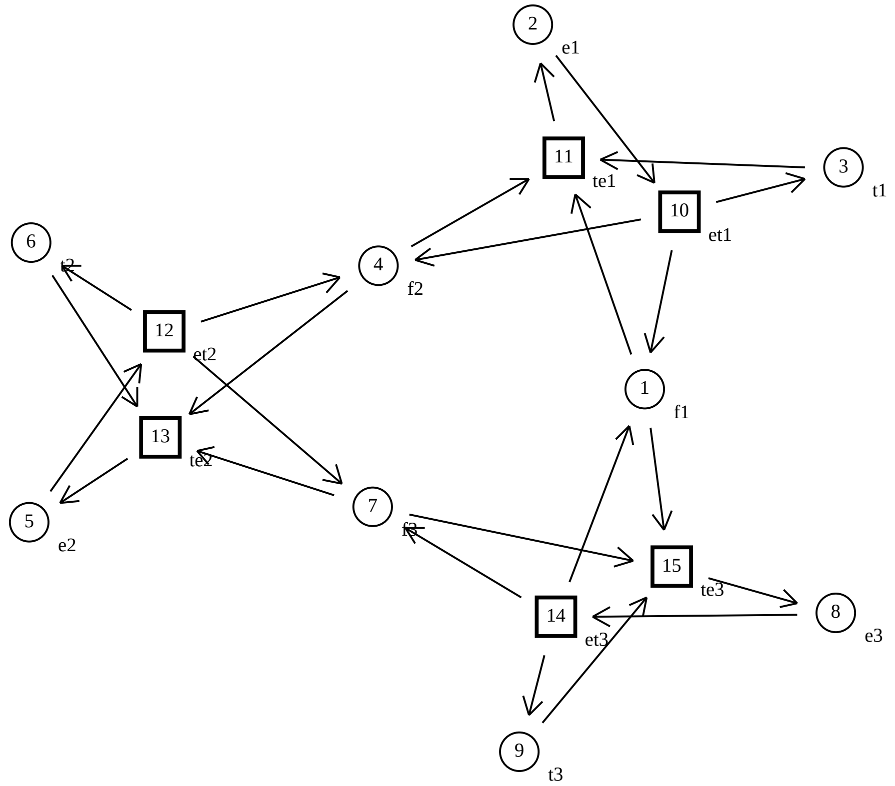
4.2 Analysis of Petri Nets

Petri nets [19] are widely used for designing and modeling concurrent and interacting processes. Since the static part of a Petri net consists of a bipartite directed graph, relational algebra in combination with RELVIEW can be used for mechanically investigating static properties like deadlocks, traps, causality, and the free choice property. See [5, 6]. But, as demonstrated in the later of these articles, even a relation-algebraic analysis of dynamic properties like reachability and liveness with RELVIEW is possible. In the following, we take two of the examples of [6] and show how to specify and implement algorithms which can be executed directly in RELVIEW.

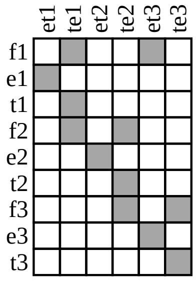
4.2.1 Petri Nets and their Relational Representation

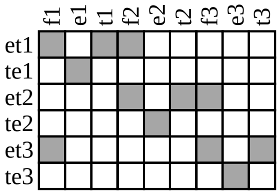
A *Petri net* is a bipartite directed graph which we represent as a pair $\mathcal{N} = (R, S)$ of relations, where $R : C \leftrightarrow E$, $S : E \leftrightarrow C$, and $C \cap E = \emptyset$. The elements of C and E are called *conditions* and *events*, respectively. In the graphical representation of $\mathcal{N}$, usually conditions are drawn as circles, events appear as squares, and the relation R (respectively S) is coded by the set of edges leading into (resp. out of) squares. A *marking* of $\mathcal{N}$ is a set of conditions which is visualized by decorating each condition in it with a bullet, called a token. Relation-algebraically, a marking is described by a vector $m : C \leftrightarrow \mathbf{1}$.

As an example, we consider the following graphical representation – produced by one of the spring embedder graph drawing algorithms of the RELVIEW system – with nine conditions $C = \{f_1, e_1, t_1, f_2, e_2, t_2, f_3, e_3, t_3\}$ and six events $E = \{et_1, te_1, et_2, te_2, et_3, te_3\}$:

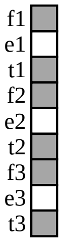


On the window of the relation editor of RELVIEW, the labeled 9×6 Boolean matrix `R` respectively the labeled 6×9 Boolean matrix `S` representing this Petri net are depicted as shown in the next two pictures:





This Petri net is a simplified description of E.W. Dijkstra's dining philosophers [9]: Three philosophers are sitting around a table, and between each two of them there is a fork as a shared resource. Each of the three philosophers $i, 1 \leq i \leq 3$, is either thinking (a token on t_i) or eating (a token on e_i). In order to start eating (event te_i) he takes the two forks f_i and f_{i+1} (respectively f_3 and f_1 if $i = 1$) provided they are free (decorated with tokens). After eating a while, he goes back into the thinking mode (event et_i) and returns the forks. Initially, all philosophers are thinking and the forks are available. This is expressed by the marking $Init = \{f_1, f_2, f_3, t_1, t_2, t_3\}$ described by the following RELVIEW vector `Init`:



The dynamic evolution of a marked Petri net is given by a simple token game which specifies the effect of events on the current marking. Given a marking $M \in 2^C$, an event $e \in E$ is currently enabled if all its predecessors but none of its successors carry a token. In this case its execution (or firing) results in a new marking $N \in 2^C$ which is obtained from the previous marking M by removing all predecessors of e and then adding all successors of e. In this way, every Petri net induces a labeled transition relation $M \xrightarrow{e} N$.

4.2.2 Computing Reachable Markings

We assume a Petri net $\mathcal{N} = (R, S)$ with conditions C and events E. Given two markings M and N, we say that N is reachable from M if and only if there is a sequence of transitions $M \xrightarrow{e_1} \ldots \xrightarrow{e_n} N$ that transforms M into N.

Since the notion of reachability is defined in terms of sequences of transitions, in the first part of our development of a relational reachability algorithm we consider a single transition from a marking M to a marking N which is caused by the execution of an event e. We have to transscribe the definition of the transition relation of a Petri net into a logical predicate. The first condition in that definition requires that M enables e which yields the formula

$$(\forall c\, R_{ce} \to c \in M) \wedge (\forall c\, S_{ec} \to c \notin M)\,.$$

Now, we represent events by points from $[E \leftrightarrow \mathbf{1}]$. Then $R\,e : C \leftrightarrow \mathbf{1}$ is the vector of the set of predecessors and $S^{\mathsf{T}}\,e : C \leftrightarrow \mathbf{1}$ is the vector of the set of successors of the event $e : E \leftrightarrow \mathbf{1}$. Furthermore, a condition $c \in C$ is a predecessor of e if and only if $(R\,e)_c$ and a successor of e if and only if $(S^{\mathsf{T}}\,e)_c$. Hence, the above formula becomes

$$(\forall c\, (R\,e)_c \to c \in M) \wedge (\forall c\, (S^{\mathsf{T}}\,e)_c \to c \notin M)\,.$$

Using the correspondences between certain kinds of logical and relation-algebraic constructions, our next aim is to replace the set-theoretic and logical symbols of this formula with relational operations and "outermost" subscripts M and N. The desired form is derived by

$$\begin{aligned}
& (\forall c\, (R\,e)_c \to c \in M) \wedge (\forall c\, (S^{\mathsf{T}}\,e)_c \to c \notin M) \\
\iff & (\forall c\, (R\,e)_c \to \varepsilon_{cM}) \wedge (\forall c\, (S^{\mathsf{T}}\,e)_c \to \overline{\varepsilon}_{cM}) \\
\iff & (\forall c\, (R\,e\,\mathsf{L})_{cN} \to \varepsilon_{cM}) \wedge (\forall c\, (S^{\mathsf{T}}\,e\,\mathsf{L})_{cN} \to \overline{\varepsilon}_{cM}) \\
\iff & (R\,e\,\mathsf{L} \setminus \varepsilon)_{NM} \wedge (S^{\mathsf{T}}\,e\,\mathsf{L} \setminus \overline{\varepsilon})_{NM} \\
\iff & ((R\,e\,\mathsf{L} \setminus \varepsilon)^{\mathsf{T}} \cap (S^{\mathsf{T}}\,e\,\mathsf{L} \setminus \overline{\varepsilon})^{\mathsf{T}})_{MN}\,,
\end{aligned}$$

where the type of the universal relation is $\mathsf{L} : \mathbf{1} \leftrightarrow 2^C$ and $\varepsilon : C \leftrightarrow 2^C$ is the membership relation on conditions.

The second condition of the labeled transition relation $M \xrightarrow{e} N$ says: If e is executed, then the new marking N results from the old marking M by replacing the predecessors of e with its successors. On account of our point representation $e : E \leftrightarrow \mathbf{1}$ of events and since thus $\overline{R\,e} : C \leftrightarrow \mathbf{1}$ is the complement of the set of predecessors of e, this is specified by

$$\forall c\, (c \in M \wedge \overline{R\,e}_c) \vee (S^{\mathsf{T}}\,e)_c \leftrightarrow c \in N\,.$$

Again, we are able to replace all the set-theoretic and predicate logic symbols with relational operations and subscripts M and N; a possible derivation is

$$\begin{aligned}
& \forall c\, (c \in M \wedge \overline{R\,e}_c) \vee (S^{\mathsf{T}}\,e)_c \leftrightarrow c \in N \\
\iff & \forall c\, (\varepsilon_{cM} \wedge \overline{R\,e}_c) \vee (S^{\mathsf{T}}\,e)_c \leftrightarrow \varepsilon_{cN} \\
\iff & \forall c\, (\varepsilon_{cM} \wedge (\overline{R\,e}\,\mathsf{L})_{cM}) \vee (S^{\mathsf{T}}\,e\,\mathsf{L})_{cM} \leftrightarrow \varepsilon_{cN} \\
\iff & \forall c\, ((\varepsilon \cap \overline{R\,e}\,\mathsf{L}) \cup S^{\mathsf{T}}\,e\,\mathsf{L})_{cM} \leftrightarrow \varepsilon_{cN} \\
\iff & \mathsf{syq}((\varepsilon \cap \overline{R\,e}\,\mathsf{L}) \cup S^{\mathsf{T}}\,e\,\mathsf{L}, \varepsilon)_{MN}\,,
\end{aligned}$$

where the types of the universal relation L and the membership relation ε are as above. Next, we can remove the subscripts M and N in the results of the last two derivations. Putting together the remainig relational terms, we get

$$\mathit{Trans}(R,S,e) = (R\,e\,\mathsf{L} \setminus \varepsilon)^{\mathsf{T}} \cap (S^{\mathsf{T}}\,e\,\mathsf{L} \setminus \overline{\varepsilon})^{\mathsf{T}} \cap \mathsf{syq}((\varepsilon \cap \overline{R\,e}\,\mathsf{L}) \cup S^{\mathsf{T}}\,e\,\mathsf{L}, \varepsilon) : 2^C \leftrightarrow 2^C$$

as a relation-algebraic description of a relation that describes all possible single transitions between markings which are caused by an execution of the event $e : E \leftrightarrow \mathbf{1}$.

Having derived a relation-algebraic description of the transition relation, we have solved the most difficult part of the reachability problem. By definition, the reachability relation

$Reach(R,S)$ on markings we have searched for is precisely the reflexive-transitive closure of the union of all transition relations:

$$Reach(R,S) = (\bigcup_{e \in \mathsf{P}(E)} Trans(R,S,e))^* : 2^C \leftrightarrow 2^C$$

Here $\mathsf{P}(E)$ denotes the set of all points from $[E \leftrightarrow \mathbf{1}]$. Also testing whether one marking can be reached from another is now trivial. If they are given as vectors $m : C \leftrightarrow \mathbf{1}$ and $n : C \leftrightarrow \mathbf{1}$, first, we produce the corresponding points $\mathsf{syq}(\varepsilon, m) : 2^C \leftrightarrow \mathbf{1}$ and $\mathsf{syq}(\varepsilon, n) : 2^C \leftrightarrow \mathbf{1}$ in the powerset (see Section 2.4.3). Then, we have the equivalence

$$n \text{ is reachable from } m \iff \mathsf{syq}(\varepsilon, m)\, \mathsf{syq}(\varepsilon, n)^{\mathsf{T}} \subseteq Reach(R,S)\,.$$

Likewise to obtain a vector describing the set $\mathcal{M}$ of all markings reachable from $m : C \leftrightarrow \mathbf{1}$ is easy. We only have to compute the relation-theoretic successors wrt. the reachability relation of the point corresponding to m:

$$ReachVector(R,S,m) = Reach(R,S)^{\mathsf{T}}\, \mathsf{syq}(\varepsilon, m) : 2^C \leftrightarrow \mathbf{1}$$

As in the case of Dedekind cuts, we can represent the elements contained in the subset $\mathcal{M}$ of 2^C described by this vector column-wise by

$$ReachRelation(R,S,m) = \varepsilon\, \mathsf{inj}(ReachVector(R,S,m))^{\mathsf{T}} : C \leftrightarrow \mathcal{M}\,.$$

If we transform the just developed relation-algebraic descriptions into the language of RELVIEW, then we obtain the following relational programs respectively functions, where we have decided to formulate `Reach` by means of the base operations `init` and `next` generating the set of events:

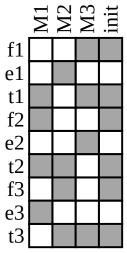
```
Trans(R,S,e)
  DECL eps, L, res
  BEG  eps = epsi(dom(R));
       L = L1n(eps);
       res = (R * e * L \ eps)^;
       res = res & (S^ * e * L \ -eps)^;
       res = res & syq((eps & -(R * e) * L) | S^ * e * L, eps)
       RETURN res
  END.

Reach(R,S)
  DECL e, res
  BEG  e = init(dom(S));
       res = Trans(R,S,e);
       WHILE -empty(next(e)) DO
         e = next(e);
         res = res | Trans(R,S,e) OD
       RETURN refl(trans(res))
  END.

ReachVector(R,S,m) = Reach(R,S)^ * syq(epsi(dom(R)),m).

ReachRelation(R,S,m) = epsi(dom(R)) * inj(ReachVector(R,S,m))^.
```

Evaluating the relational term `ReachRelation(R,S,Init)` with RELVIEW, where `R` and `S` are the relations of the philosophers net and `Init` describes its initial marking *Init*, produces the subsequent column-wise representation of the four markings reachable from `Init`, where labels are added for illustration purposes:



The last column of this 9×4 Boolean matrix describes the initial marking *Init*, where all philosophers are thinking. As can also be seen from this Boolean matrix, besides this marking three different markings are reachable from the initial one. Each of them corresponds to one of the first three columns and expresses that exactly one philosopher eats and the others think.

4.2.3 Liveness of Markings

There are several notions of a marking of a Petri net to be live, see [17]. The following version seems to be preferred in the literature: Given a Petri net $\mathcal{N} = (R, S)$, an event $e \in E$ is said to be dead under a marking $M \in 2^C$ if there is no marking $N \in 2^C$ reachable from M which enables e. A marking $M \in 2^C$ is called live if for all markings $N \in 2^C$ reachable from M and all events $e \in E$ we have that e is not dead under N.

Let again $\varepsilon : C \leftrightarrow 2^C$ be the membership relation on conditions. We start our development of a relation-algebraic description of liveness with

$$(\forall c\; R_{ce} \to c \in M) \land (\forall c\; S_{ec} \to c \notin M)$$

which specifies that the marking M enables the event e. In contrast with Section 4.2.2, however, we do not represent events by points in the relational sense. This allows the following derivation which replaces the set-theoretic and predicate logic symbols with relational operations and the subscripts M and e:

$$\begin{aligned} & (\forall c\; R_{ce} \to c \in M) \land (\forall c\; S_{ec} \to c \notin M) \\ \iff\; & (\forall c\; R^{\mathsf{T}}_{ec} \to \varepsilon^{\mathsf{T}}_{Mc}) \land (\forall c\; S_{ec} \to \overline{\varepsilon^{\mathsf{T}}}_{Mc}) \\ \iff\; & (\varepsilon^{\mathsf{T}} \,/\, R^{\mathsf{T}})_{Me} \land (\overline{\varepsilon^{\mathsf{T}}} \,/\, S)_{Me} \\ \iff\; & ((\varepsilon^{\mathsf{T}} \,/\, R^{\mathsf{T}}) \cap (\overline{\varepsilon^{\mathsf{T}}} \,/\, S))_{Me} \end{aligned}$$

Now, the subscripts M and e can be removed from the last formula, yielding

$$\mathit{Enable}(R, S) = (\varepsilon^{\mathsf{T}} \,/\, R^{\mathsf{T}}) \cap (\overline{\varepsilon^{\mathsf{T}}} \,/\, S) : 2^C \leftrightarrow E$$

as relation-algebraic description of the enabling relation. Combining it with the reachability relation $\mathit{Reach}(R, S)$ derived in the last section, we have that an event $e \in E$ is dead under a marking $M \in 2^C$ if and only if

$$\neg\exists\, N \;\; \mathit{Reach}(R,S)_{MN} \wedge \mathit{Enable}(R,S)_{Ne}\,.$$

So the set of all such pairs M, e relation-algebraically is specified by

$$\mathit{Dead}(R,S) = \overline{\mathit{Reach}(R,S)\,\mathit{Enable}(R,S)} \;:\; 2^C \leftrightarrow E\,.$$

To specify liveness in predicate logic, finally, we use the reachability relation $\mathit{Reach}(R,S)$ again, but now in combination with $\mathit{Dead}(R,S)$. We get that a marking M is live if and only if the formula

$$\forall\, N\ \forall\, e\ \mathit{Reach}(R,S)_{MN} \rightarrow \neg\mathit{Dead}(R,S)_{Ne}$$

holds. In this case, the replacement of the set-theoretic and predicate logic symbols with relational operations and the subscript M follows from

$$\begin{array}{cl}
 & \forall\, N\ \forall\, e\ \mathit{Reach}(R,S)_{MN} \rightarrow \neg\mathit{Dead}(R,S)_{Ne} \\
\Longleftrightarrow & \forall\, N\ \mathit{Reach}(R,S)_{MN} \rightarrow \neg\exists\, e\ \mathit{Dead}(R,S)_{Ne} \\
\Longleftrightarrow & \neg\exists\, N\ \mathit{Reach}(R,S)_{MN} \wedge \exists\, e\ \mathit{Dead}(R,S)_{Ne} \\
\Longleftrightarrow & \neg\exists\, N\ \mathit{Reach}(R,S)_{MN} \wedge (\mathit{Dead}(R,S)\,\mathsf{L})_N \\
\Longleftrightarrow & \overline{\mathit{Reach}(R,S)\,\mathit{Dead}(R,S)\,\mathsf{L}}_{M}\,,
\end{array}$$

using an universal vector $\mathsf{L} : E \leftrightarrow \mathbf{1}$. Finally, a removal of the subscript M yields

$$\mathit{LiveVector}(R,S) = \overline{\mathit{Reach}(R,S)\,\mathit{Dead}(R,S)\,\mathsf{L}} \;:\; 2^C \leftrightarrow \mathbf{1}$$

as the vector which describes the set $\mathcal{L}$ of all markings which are live and

$$\mathit{LiveRelation}(R,S,m) = \varepsilon\,\mathsf{inj}(\mathit{LiveVector}(R,S,m))^{\mathsf{T}} : C \leftrightarrow \mathcal{L}$$

as the column-wise representation of the set $\mathcal{L}$. To avoid repeated evaluations of relational terms, in the following RELVIEW implementations of *Enable*, *LiveVector*, and *LiveRelation* we have used relational programs instead of relational functions.

```
Enable(R,S)
  DECL eps
  BEG  eps = epsi(dom(R))
       RETURN (eps^ / R^) & (-eps^ / S)
  END.

LiveVector(R,S)
  DECL reach, dead
  BEG  reach = Reach(R,S);
       dead = -(reach * Enable(R,S))
       RETURN -dom(reach * dead)
  END.

LiveRelation(R,S) = epsi(dom(R)) * inj(LiveVector(R,S))^.
```

As an example, we consider again the relations `R` and `S` of the philosophers net. If we evaluate the relational term `LiveRelation(R,S)` with RELVIEW and add labels to the result, we obtain the following 9×8 Boolean matrix:

	M1	M2			M3	init		
f1	□	□	□	□	■	■	■	■
e1	□	■	■	■	□	□	□	■
t1	■	□	□	□	■	■	■	□
f2	■	□	□	■	□	■	■	□
e2	□	□	■	□	■	□	■	■
t2	■	■	□	■	□	■	□	□
f3	□	■	□	■	□	■	□	■
e3	■	□	■	■	□	□	■	□
t3	□	■	□	□	■	■	□	■

From the columns 1, 2, 5, and 6 we see that every marking reachable from the initial marking *init* is live, i.e.,

$$ReachVector(R, S, init) \subseteq LiveVector(R, S)$$

holds, a test which can easily be verified with RELVIEW. This means that the marked philosophers net $\mathcal{N} = (R, S, Init)$ is live. There are four more live markings, but none of them corresponds to a "real" state in a philosopher's dinner. For example, the marking $\{e_1, e_2, e_3\}$ depicted in the third column describes the impossible situation that each philosopher is eating.

4.3 Solving Graph-Theoretic Problems

Graphs are the most common abstract structure in computer science. There are various types of graphs which appear in the literature, e.g., directed graphs, undirected graphs, simple graphs, hypergraphs, bipartite graphs. If one does not allow edges to be independent mathematical objects but consider them as pairs of nodes, then this kind of graphs (often called 1-graphs) and relations are very closely related. In principle, such a graph $g = (X, R)$ is given by its associated relation $R : X \leftrightarrow X$ on the set X of nodes. Many applications require algorithms that operate on graphs, since any system that consists of discrete states or sites and connections between them can be modeled by a graph. In this section, we show how a relation-algebraic approach to graph theory can be used to develop such algorithms. Most of the examples are taken from [5, 7].

4.3.1 Computing Kernels

Suppose a graph $g = (X, R)$. A set $a \in 2^X$ of nodes is said to be absorbant if from every node outside of it there is at least one edge leading into it, a property which is described by

$$\forall\, x\ x \notin a \rightarrow (\exists\, y\ y \in a \wedge R_{xy}).$$

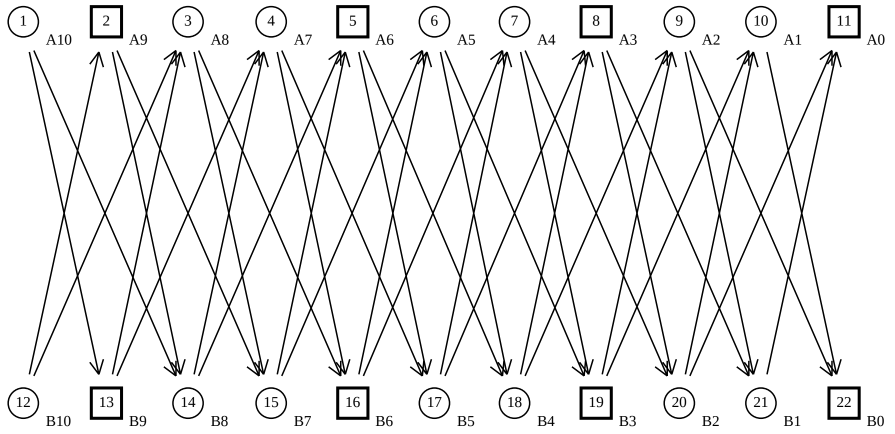
Furthermore, a set $s \in 2^X$ of nodes is called stable if no two nodes of it are related via the relation R. This specific situation is characterized by

$$\forall\, x\ x \in s \rightarrow (\exists\, y\ y \in s \rightarrow \overline{R}_{xy}).$$

And, finally, a *kernel* of the graph g is a set of nodes which is at the same time absorbant and stable.

The concept of a kernel plays an import rôle in combinatorial games, since kernels in game graphs correspond to winning strategies. To explain this, we consider the following well-known game: From a pile of, say, 10 matches two players A and B may alternately take one or two matches, which is called a move. If some player has to move but cannot do this because the pile is empty, then he loses.

Represented in the RELVIEW system as a circuit-free, bipartite graph, this matches game looks at follows:



Each node of this graph stands for a specific situation which can occur during the game. If it is labeled with Ai (respectively Bi), then this means that the pile consists of i matches and player A (respectively B) has to move. Hence, the possible moves are represented by the edges and the terminal nodes with labels A0 respectively B0 stand for the situations that player A respectively B loses.

The above game graph has exactly one kernel the nodes of which are drawn as squares. Its knowledge provides the player moving from some situation outside it with the winning strategy "move into the kernel".

To develop a relation-algebraic description of the set of all kernels of a graph $g = (X, R)$ as a vector from $[2^X \leftrightarrow \mathbf{1}]$, we start with the first of the above two formulae, saying that a set a of nodes is absorbant. Using the correspondences between logical and relation-algebraic constructions, then we transform it as follows:

$$
\begin{aligned}
& \forall\, x\; x \notin a \rightarrow (\exists\, y\; y \in a \wedge R_{xy}) \\
\Longleftrightarrow\; & \forall\, x\; \varepsilon_{xa} \vee (\exists\, y\; \varepsilon_{ya} \wedge R_{xy}) \\
\Longleftrightarrow\; & \forall\, x\; \varepsilon_{xa} \vee (R\,\varepsilon)_{xa} \\
\Longleftrightarrow\; & \forall\, x\; (\varepsilon \cup R\,\varepsilon)_{xa} \\
\Longleftrightarrow\; & (\overline{\overline{\varepsilon \cup R\,\varepsilon}} \setminus \mathsf{O})_a
\end{aligned}
$$

This derivation introduces the membership relation $\varepsilon : X \leftrightarrow 2^X$ and the empty vector $\mathsf{O} : X \leftrightarrow \mathbf{1}$. For the removal of the universal quantification using a right residual construction in the last step, see Section 2.3.2. Now, we remove the subscript a in $(\overline{\overline{\varepsilon \cup R\,\varepsilon}} \setminus \mathsf{O})_a$ and obtain, thus,

$$
\mathit{AbsorbVector}(R) = \overline{\overline{\varepsilon \cup R\,\varepsilon}} \setminus \mathsf{O} : 2^X \leftrightarrow \mathbf{1}
$$

as relation-algebraic description of the vector of the absorbant sets. An analogous derivation shows for the vector of the stable sets the relation-algebraic description

$$StableVector(R) = \overline{(\varepsilon \cap R\,\varepsilon) \setminus \mathsf{O}} : 2^X \leftrightarrow \mathbf{1}\,,$$

where the types of the membership relation and the empty vector are as in the case of the function $AbsorbVector$. Now, the vector describing the elements of 2^X which are kernels of g is given as intersection

$$KernelVector(R) = AbsorbVector(R) \cap StableVector(R) : 2^X \leftrightarrow \mathbf{1}\,.$$

If this vector is non-empty, i.e., the graph $g = (X, R)$ has at least one kernel, then the column-wise representation of the set $\mathcal{K}$ of all kernels of g is

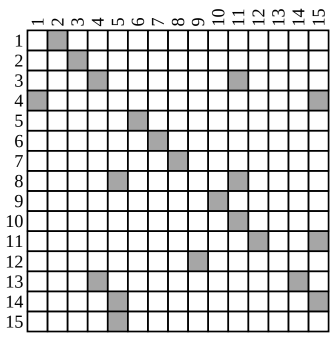
$$KernelRelation(R) = \varepsilon\,\mathsf{inj}(KernelVector(R))^{\mathsf{T}} : X \leftrightarrow \mathcal{K}\,.$$

In each of the relation-algebraic descriptions $AbsorbVector(R)$ and $StableVector(R)$ the membership relation $\varepsilon : X \leftrightarrow 2^X$ on nodes appears twice. In order to avoid repeated evaluations of the corresponding relational term `epsi(dom(R))`, in the following RELVIEW versions of $KernelVector$ and $KernelRelation$ we implement the first function by a relational program:

```
KernelVector(R)
  DECL AbsorbVector(R,e,O) = -(e | R * e) \ O;
       StableVector(R,e,O) = (e & R * e) \ O;
       eps
  BEG  eps = epsi(dom(R))
       RETURN AbsorbVector(R,eps,On1(R)) & StableVector(R,eps,On1(R))
  END.

KernelRelation(R) = epsi(dom(R)) * inj(KernelVector(R))^.
```

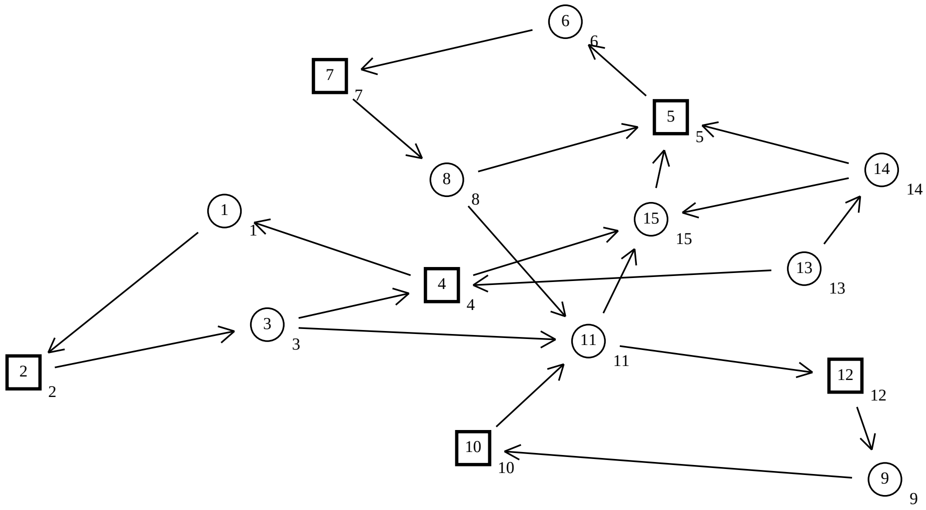
Now, let us consider a concrete example for computing kernels. We assume the node set of our example graph $g = (X, R)$ to consist of the natural numbers from 1 through 15. The following picture shows the relation of g as presented on the window of the relation editor of RELVIEW as labeled 15×15 Boolean matrix `R`:



To produce the column-wise representation of the four kernels of the graph given by this relation using RELVIEW, we have to evaluate the relational term `KernelRelation(R)` and obtain then – after an appropriate labeling of the rows and columns of the result – the following 15×4 Boolean matrix:

	K1	K2	K3	K4
1			■	■
2	■	■		
3			■	■
4	■	■		
5	■	■		■
6			■	
7	■	■		■
8			■	
9		■		
10	■		■	■
11		■		
12	■		■	■
13			■	■
14				
15			■	

In the next picture of this section we show the relation `R` as a directed graph, produced by a spring embedder graph drawing algorithm of RELVIEW.



In this directed graph also the kernel $K_1 = \{2,4,5,7,10,12\}$ of g, described by the first column of the above kernel relation, is indicated by squares.

Computing the kernels of a graph in the way described above, frequently may be fairly inefficient. This is not the fault of relational algebra, as the problem is in fact NP-complete as shown in [12]. For specific classes of graphs, however, a combination of relational algebra and fixpoint theory (see [23, 18]) allows the development of efficient algorithms for computing kernels; see [20] for example. In the following, we present such a case.

We consider again a graph $g = (X, R)$. In contrast with the first enumeration approach, however, we consider now a single kernel as a vector $v : X \leftrightarrow \mathbf{1}$. In doing so, the logical formulae defining absorbant and stable sets become

$$\forall x\ \overline{a}_x \to (\exists y\ a_y \wedge R_{xy}) \qquad \forall x\ s_x \to (\exists y\ s_y \to \overline{R}_{xy}).$$

Translating these formulae into a notation without components, we get the two inclusions $\overline{a} \subseteq R\,a$ and $s \subseteq \overline{R\,s}$. As a consequence, a vector $k : X \leftrightarrow \mathbf{1}$ describes a kernel of g if and only if $\overline{k} = R\,k$, i.e., if and only if it is a fixpoint of the function

$$\tau : [X \leftrightarrow \mathbf{1}] \to [X \leftrightarrow \mathbf{1}] \qquad \tau(v) = \overline{R\,v}\,.$$

This function is antitone (order-reversing), so A. Tarski's well-known fixpoint theorem for monotone functions on complete lattices (see [23]) cannot be applied. We therefore study the fixpoints of its square

$$\tau^2 : [X \leftrightarrow \mathbf{1}] \to [X \leftrightarrow \mathbf{1}] \qquad \tau^2(v) = \tau(\tau(v)) = \overline{R\,\overline{R\,v}}$$

which is monotone. Suppose $m_{\tau^2} : X \leftrightarrow \mathbf{1}$ and $M_{\tau^2} : X \leftrightarrow \mathbf{1}$ to denote the least resp. greatest fixpoint of τ^2. Then we have for each kernel $k : X \leftrightarrow \mathbf{1}$ of g that

$$\mathsf{O} \subseteq \tau^2(\mathsf{O}) \subseteq \tau^4(\mathsf{O}) \subseteq \ldots \subseteq m_{\tau^2} \subseteq k \subseteq M_{\tau^2} \subseteq \ldots \subseteq \tau^4(\mathsf{L}) \subseteq \tau^2(\mathsf{L}) \subseteq \mathsf{L}\,.$$

Also the two equations $\tau(m_{\tau^2}) = M_{\tau^2}$ and $\tau(M_{\tau^2}) = m_{\tau^2}$ easily can be shown. Hence, if the function τ^2 has exactly one fixpoint, which is equivalent to $M_{\tau^2} \subseteq \tau(M_{\tau^2})$ or to $\tau(m_{\tau^2}) \subseteq m_{\tau^2}$, then g has precisely one kernel.

Using this fact, for instance, it can be shown that a progressively finite graph, i.e., a graph in which all paths have finite lengths, has exactly one kernel. When specifying progressive finiteness of $g = (X, R)$ with relation-algebraic means, we obtain that

$$v \subseteq R\,v \implies v = \mathsf{O} \tag{$*$}$$

for all vectors $v : X \leftrightarrow \mathbf{1}$. Compare [5, 7]. Now, we use the Schröder equivalences and get $R^{\mathsf{T}}\,M_{\tau^2} \subseteq R\,M_{\tau^2}$ from $M_{\tau^2} \subseteq \tau^2(M_{\tau^2})$. Next, we have

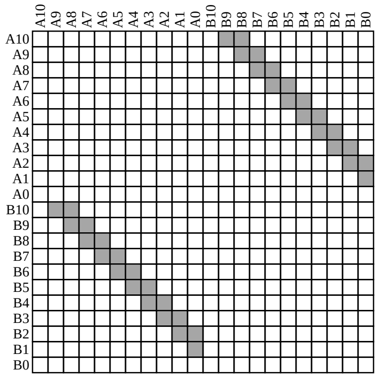
$$\begin{array}{rcll} R\,M_{\tau^2} \cap M_{\tau^2} & \subseteq & (R \cap M_{\tau^2}\,{M_{\tau^2}}^{\mathsf{T}})\,(M_{\tau^2} \cap R^{\mathsf{T}}\,M_{\tau^2}) & \text{Dedekind rule} \\ & \subseteq & R\,(M_{\tau^2} \cap R^{\mathsf{T}}\,M_{\tau^2}) & \text{Monotonicity} \\ & \subseteq & R\,(M_{\tau^2} \cap R\,M_{\tau^2}) & \text{see above.} \end{array}$$

In combination with the relational description of progressively finiteness by implication $(*)$, we obtain the equation $R\,M_{\tau^2} \cap M_{\tau^2} = \mathsf{O}$ i.e., the inclusion $M_{\tau^2} \subseteq \tau(M_{\tau^2})$. Hence, the function τ^2 and, therefore, also the function τ have precisely one fixpoint.

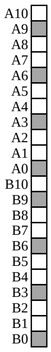
For the set of nodes being finite, we have that a directed graph is progressively finite if and only if it is circuit-free. Using the RELVIEW system, therefore, we can compute the only kernel of a finite, circuit-free graph by the following relational program:

```
KernelNoetherian(R)
  DECL k, v
  BEG  k = On1(R);
       v = -(R * -(R * k));
       WHILE -eq(k,v) DO
         k = v;
         v = -(R * -(R * k))
         OD
       RETURN k
  END.
```

This program computes the kernel as the limit of the chain $\mathsf{O} \subseteq \tau^2(\mathsf{O}) \subseteq \tau^4(\mathsf{O}) \subseteq \ldots$ for the least fixpoint of τ^2. Its run time complexity is $O(n^3)$, where n is the cardinality of the set of nodes. If it is applied to the 22×22 Boolean matrix



which is the matrix representation of the relation `R` of the matches game, then we obtain the following 22×1 Boolean vector



This vector exactly describes the marked nodes of the graph representation of `R` given at the beginning of this section.

4.3.2 Algorithms for Computing Transitive Closures

Assume $g = (X, R)$ to be a graph and we have to test whether there is a path between two given nodes. If many of such questions will be asked about the same graph and response time is critical, then it is a good idea to compute the transitive closure R^+ of the relation R once and for all, since then subsequently queries can be answered by simple look-up. This works because a node $y \in X$ can be reached from another node $x \in X$ just when $(R^+)_{xy}$ holds.

To obtain a first algorithm for transitive closures, recall that R^+ is the smallest transitive relation which contains the relation R. Therefore, we have

$$R^+ = \bigcap\{Q : R \subseteq Q, Q\,Q \subseteq Q\} = \bigcap\{Q : R \cup Q\,Q \subseteq Q\}\,.$$

Looking to A. Tarskis fixpoint theorem for monotonic functions on complete lattices [23], from the third expression of this equation we obtain R^+ as the least fixpoint μ_σ of the monotone function

$$\sigma : [X \leftrightarrow X] \to [X \leftrightarrow X] \qquad\qquad \sigma(Q) = R \cup Q\,Q\,,$$

on relations, i.e., as the limit of the chain $\mathsf{O} \subseteq \sigma(\mathsf{O}) \subseteq \sigma^2(\mathsf{O})) \subseteq \ldots$ which is finite provided the node set X of the graph is finite. With the RELVIEW system, this limit can be computed by the following relational program:

```
TransCl(R)
  DECL Q, S
  BEG  Q = O(R);
       S = R;
       WHILE -eq(Q,S) DO
         Q = S;
         S = R | Q * Q
         OD
       RETURN Q
  END.
```

Obviously, the run time complexity of `TransCl` is $O(n^3 \log n)$, where n is the cardinality of the set of nodes.

A more efficient algorithm for computing transitive closures was proposed by S. Warshall [25]. It relies on a clever problem generalization similar forms of which occur in many transformational developments. Relational algebra allows us to capture this idea in a concise calculation, completely avoiding informal ad-hoc arguments about the existence of paths and intermediate nodes. In the following, we show how formally to develop Warshall's transitive closure algorithm from a specification by combining relational algebra and well-known techniques that aid the construction of imperative programs. These techniques can be traced back to [10].

Our problem is to find a relational program with a parameter R of type $[X \leftrightarrow X]$ and a relation-valued local variable Q of the same type, such that after the execution of its body the postcondition (specification)

$$post(R, Q) \;:\Longleftrightarrow\; Q = R^+$$

holds[1]. As a generalization of this postcondition, we consider for a vector $v : X \leftrightarrow \mathbf{1}$ the formula

$$Inv(R, Q, v) \;:\Longleftrightarrow\; Q = (R\,\mathsf{I}_v)^*\,R\,,$$

where $\mathsf{I}_v = \mathsf{I} \cap v\,v^{\mathsf{T}}$ is the partial identity given by v. This formula is the relation-algebraic description of the fact that Q consists of the pairs $\langle x, y\rangle \in X \times X$ of nodes for which there exists a path from x to y in $g = (X, R)$ the "inner" nodes of which are from the set described by v. Hence, we suppose the relational program to be developed to contain in addition to Q a vector-valued local variable v.

[1]Usually, a specification of an imperative program consists of a postcondition and a precondition, but in our specific case the latter one may be assumed as formula *True*.

From $(R\,\mathsf{I}_{\mathsf{L}})^*\,R = R^*\,R = R^+$ we obtain that $\mathit{Inv}(R,Q,v)$ and $v = \mathsf{L}$ imply the post-condition $\mathit{post}(R,Q)$. Guided by this fact, we choose $\mathit{Inv}(R,Q,v)$ as invariant and $v = \mathsf{L}$ as negation of the guard of the while-loop and look – in the notation of RELVIEW – for a relational program of the following form:

```
Warshall(R)
  DECL Q, v
  BEG  ≫initialization≪;
       WHILE -eq(v,L(v)) DO
         ≫loop body≪
         OD
       RETURN Q
  END.
```

It remains to find an initialization which establishes the invariant, and a loop body which maintains it.

Due to the equation $R = \mathsf{O}^*\,R = (R\,\mathsf{I}_{\mathsf{O}})^*\,R$ we have $\mathit{Inv}(R,R,\mathsf{O})$ and it seems reasonable to choose, again in RELVIEW notation, the assignments

```
Q = R; v = On1(R)
```

as initialization. Since then the while-loop starts with the empty vector, a natural choice for the variant function is $v \mapsto v \cup \mathsf{point}(\,\overline{v}\,)$ as this ensures its termination if the graph is finite. For the following calculations, we introduce p as shorthand for $\mathsf{point}(\,\overline{v}\,)$.

Assume $v \neq \mathsf{L}$ and the invariant $\mathit{Inv}(Q,R,v)$. Then the point p is defined. To work out the loop body, we will use the star decomposition rule $(S \cup T)^* = (S^*\,T)^*\,S^*$ and that $(S\,w\,w^{\mathsf{T}})^* = \mathsf{I} \cup S\,w\,w^{\mathsf{T}}$ for any vector w. A proof of the first equation can be found in [21], the latter equation immediately follows from the fact that $S\,w\,w^{\mathsf{T}}$ is transitive. Next, we have for the partial identities I_v and $\mathsf{I}_{v\cup p}$ the relationship

$$
\begin{aligned}
\mathsf{I}_v \cup p\,p^{\mathsf{T}} &= (\mathsf{I} \cap v\,v^{\mathsf{T}}) \cup p\,p^{\mathsf{T}} \\
&= \mathsf{I} \cap (v\,v^{\mathsf{T}} \cup p\,p^{\mathsf{T}}) && \text{as } p\,p^{\mathsf{T}} \subseteq \mathsf{I} \text{ due to } (\mathrm{E}_2) \\
&= \mathsf{I} \cap ((v \cup p)\,(v \cup p)^{\mathsf{T}}) && (\mathrm{E}_1) \text{ implies } v\,p^{\mathsf{T}} \subseteq \overline{\mathsf{I}} \text{ and } p\,v^{\mathsf{T}} \subseteq \overline{\mathsf{I}} \\
&= \mathsf{I}_{v\cup p}
\end{aligned}
$$

as expected. Using it, we are able to derive the equation

$$
\begin{aligned}
(R\,\mathsf{I}_{v\cup p})^*\,R &= (R\,(\mathsf{I}_v \cup p\,p^{\mathsf{T}}))^*\,R \\
&= (R\,\mathsf{I}_v \cup R\,p\,p^{\mathsf{T}})^*\,R \\
&= ((R\,\mathsf{I}_v)^*\,R\,p\,p^{\mathsf{T}})^*\,(R\,\mathsf{I}_v)^*\,R && \text{star decomposition rule} \\
&= (\mathsf{I} \cup (R\,\mathsf{I}_v)^*\,R\,p\,p^{\mathsf{T}})\,(R\,\mathsf{I}_v)^*\,R && (S\,w\,w^{\mathsf{T}})^* = \mathsf{I} \cup S\,w\,w^{\mathsf{T}} \\
&= (R\,\mathsf{I}_v)^*\,R \cup (R\,\mathsf{I}_v)^*\,R\,p\,p^{\mathsf{T}}\,(R\,\mathsf{I}_v)^*\,R \\
&= Q \cup Q\,p\,p^{\mathsf{T}}\,Q && \text{assumption } \mathit{Inv}(Q,R,v),
\end{aligned}
$$

which shows that also the formula $\mathit{Inv}(R, Q \cup Q\,p\,p^{\mathsf{T}}\,Q, v \cup p)$ holds. As a consequence, the following loop body in RELVIEW notation maintains the invariant:

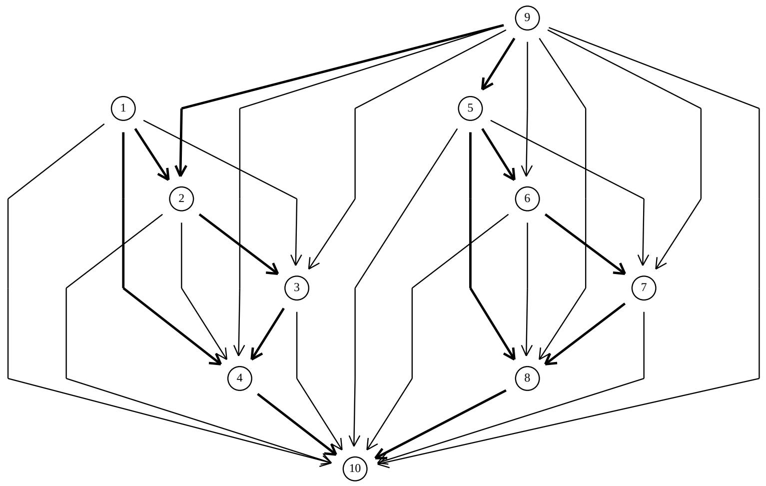
```
Q = Q | (Q * point(-v)) * (point(-v)^ * Q); v = v | point(-v)
```

Completing now the above program fragment by the initialization and the loop body just calculated and introducing after that an additional variable `p` to avoid multiple evaluations of the relational term `point(-v)`, we obtain the following relational program for computing the transitive closure of a relation:

```
Warshall(R)
  DECL Q, v, p
  BEG  Q = R;
       v = On1(R);
       WHILE -eq(v,L(v)) DO
         p = point(-v);
         Q = Q | (Q * p) * (p^ * Q);
         v = v | p
         OD
       RETURN Q
  END.
```

In the second assignment of the loop body, we have both products `Q*p` and `p^*Q` put in parentheses so that the new value of the local variable `Q` in each turnaround of the while-loop can be computed in time $O(n^2)$, where n is the number of nodes of the graph. Hence, the relational program `Warshall` runs in time $O(n^3)$.

As a concrete RELVIEW application, we present the following directed graph produced by the layer graph drawing algorithm of the system:



This directed graph represents the transitive closure of that relation which is depicted as a sub-graph by the boldface edges.

4.3.3 An Algorithm for Finding Cutnodes

We call a graph $g = (X, R)$ *simple* if its relation $R : X \leftrightarrow X$ is symmetric ($R^\mathsf{T} \subseteq R$) and irreflexive ($R \subseteq \overline{\mathsf{I}}$). A node $x \in X$ then is said to be a *cutnode* (or articulation node) of g if the sub-graph generated by the set $X \setminus \{x\}$ contains more connected components than g. This concept serves for determining "how tightly" a graph is connected. It is important in many practical applications of graph theory, e.g., in transport networks.

Given a simple graph $g = (X, R)$, we want to develop a relation-algebraic description of the vector of all cutnodes. A little reflection shows that a node $x \in X$ is a cutnode if and only if it cannot "bypassed", i.e., there exist different nodes $y \in X \setminus \{x\}$ and $z \in X \setminus \{x\}$ such that each path from y to z contains x. Now, we interpret x as a relational point $p : X \leftrightarrow \mathbf{1}$ and consider the relation $(\mathsf{inj}(\,\overline{p}\,)\, R\, \mathsf{inj}(\,\overline{p}\,)^{\mathsf{T}})^{+} \cap \overline{\mathsf{I}}$ which relates a node $y \in X \setminus \{x\}$ to a node $z \in X \setminus \{x\}$ if and only if $y \neq z$ and there exists a path from y to z in g which does not contain x. Obviously, it is included in $\mathsf{inj}(\,\overline{p}\,)\, R^{+}\, \mathsf{inj}(\,\overline{p}\,)^{\mathsf{T}} \cap \overline{\mathsf{I}}$, since this relation relates $y \in X \setminus \{x\}$ to $z \in X \setminus \{x\}$ if and only if $y \neq z$ and there exists a path from y to z in g. The reverse inclusion is equivalent to the fact that the node x can be bypassed. As a consequence, we have

$$(\mathsf{inj}(\,\overline{p}\,)\, R\, \mathsf{inj}(\,\overline{p}\,)^{\mathsf{T}})^{+} \cap \overline{\mathsf{I}} \neq \mathsf{inj}(\,\overline{p}\,)\, R^{+}\, \mathsf{inj}(\,\overline{p}\,)^{\mathsf{T}} \cap \overline{\mathsf{I}} \iff p \text{ is a cutnode}\,.$$

Now, we use $\mathsf{L} : \mathbf{1} \leftrightarrow \mathbf{1}$ and $\mathsf{O} : \mathbf{1} \leftrightarrow \mathbf{1}$ as truth values (see Section 2.3.5), the equality test eq, and a function $\mathit{Del}(R, p) = \mathsf{inj}(\,\overline{p}\,)\, R\, \mathsf{inj}(\,\overline{p}\,)^{\mathsf{T}}$ for deleting from R all edges which are incident with the node described by p. Then, we get from the above

$$\mathit{IsCut}(R, p) = \overline{\mathsf{eq}(\mathit{Del}(R, p)^{+} \cap \overline{\mathsf{I}}, \mathit{Del}(R^{+}, p) \cap \overline{\mathsf{I}})} : \mathbf{1} \leftrightarrow \mathbf{1}$$

as relation-algebraic test of a point $p : X \leftrightarrow \mathbf{1}$ to be a cutnode of the graph $g = (X, R)$. The relation-algebraic description of the vector of cutnodes we are searching for follows immediately from this test. We use that the node set X in the set-theoretic sense is isomorphic to the disjoint union $\sum_{x \in X} \mathbf{1}$ of $|X|$ copies of the specific singleton set $\mathbf{1}$. Identifying X and this disjoint union, we then get the vector of cutnodes as

$$\mathit{CutVector}(R) = \sum_{p \in \mathsf{P}(X)} \mathit{IsCut}(R, p) : X \leftrightarrow \mathbf{1}\,,$$

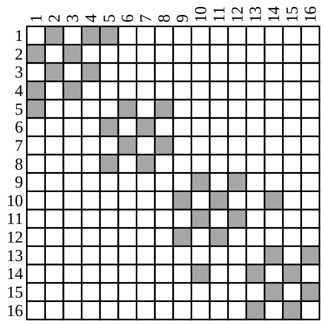
i.e., as the $|X|$-ary relational direct sum (for the binary case, see Section 2.4.2) of the truth values $\mathit{IsCut}(R, p)$, where p ranges over all points from $[X \leftrightarrow \mathbf{1}]$.

To implement *CutVector* in RELVIEW, we use the base operation for binary direct sum in combination with the base operations for generating sets and a while-loop to compute finite direct sums. This leads to the following relational program:

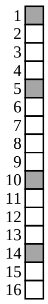
```
CutVector(R)
  DECL Ipa(R) = R & -I(R);
       Del(R,v) = inj(-v) * R * inj(-v)^;
       IsCut(R,p) = -eq(Ipa(trans(Del(R,p))),Ipa(Del(trans(R),p)));
       c, p
  BEG  p = init(Ln1(R));
       c = IsCut(R,p);
       WHILE -empty(next(p)) DO
         p = next(p);
         c = c + IsCut(R,p)
         OD
       RETURN c
  END.
```

Since transitive closures can be computed in cubic time, its run time complexity is $O(n^4)$, where n is the cardinality of the set of nodes.

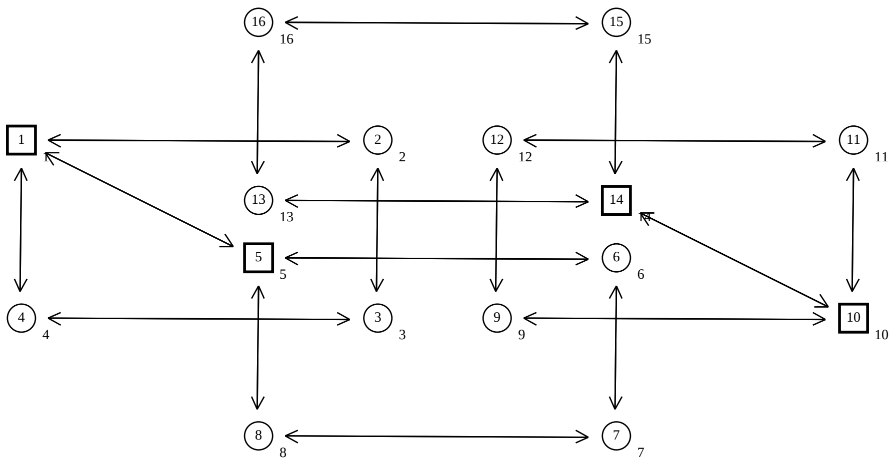
The next picture shows a symmetric and irreflexive relation `R` as a 16×16 Boolean matrix which rows and colums are labeled by the numbers from 1 through 16.



If we evaluate the relational term `CutVector(R)` with RELVIEW and after that add the row labels of `R` also to the rows of the result, on the window of the relation editor we see the following 16×1 Boolean vector:



From this vector we see that the graph corresponding to `R` has four cutnodes, viz. 1, 5, 10, and 14. In the following picture, the drawing of which was supported by the grid facility of the RELVIEW graph editor, these nodes are drawn as squares.



We will close this section with a remark on the concept dual to cutnodes. Of course, if one removes an edge instead a node, then the number of components may increase, too.

In this case, the edge is called a *bridge.* Since an edge is a bridge if and only if it cannot bypassed, it is obvious that our approach for finding the cutnodes of a simple graph can also be used for computing its bridge relation.

5 Concluding Remarks

In this report we have given a description of the computer system RELVIEW inclusive a user's manual and some examples and have also informed about the theoretical background. Besides the experiments described in the last section, a lot of other case studies have been performed with the present RELVIEW system or its predecessors. These include, for example, further graph-theoretic questions and algorithms (see [24, 15, 27]) or relational semantics (see [26]). At Kiel University, RELVIEW was and is also applied in education, i.e., in lectures and seminars.

It turns out that RELVIEW is a good tool for the interactive manipulation of relations and supports many different prototyping tasks within nearly all stages of a development of a relational program. Its real attraction is its flexibility since this property allows to experiment with new relational concepts as well as relational specifications and programs while avoiding unnecessary overhead.

Let us close with a few remarks on further developments concerning RELVIEW. Of course, a main improvement is possible in the graph layout. Presently, five different graph drawing algorithms are available. Here we plan to include further facilities for an aesthetic layout of graphs, e.g., planar drawing or orthogonal grid drawing. Another work to be done in the future concerns the error messages of the parser. In the present RELVIEW version, they are not always as helpful as they should be. E.g., when reading a syntactically faulty program from a file, the line number indicating the error is often inaccurate. Since no further hint is given, this may not be very helpful. It is planned to improve this in a future RELVIEW version. A third future extension of RELVIEW concerns the interface with other systems. E.g., an interface to the relational formula manipulation system and proof checker RALF (see [13]) is planned. Since the xrv-files created on a Sun SPARC station with Solaris and the xrv-files created on a Linux system are not interchangable, presently, we work on tools for converting relations and graphs contained in a xrv-file into ASCII format and vice versa. Besides data transfer between Solaris and Linux, this also allows to produce big relations and graphs to be manipulated within RELVIEW using a conventional programming language.

References

[1] Abold-Thalmann H., Berghammer R., Schmidt G.: Manipulation of concrete relations: The RELVIEW-system. Report Nr. 8905, Fakultät für Informatik, Universität der Bundeswehr München (1989)

[2] Berghammer R., Zierer H.: Relational algebraic semantics of deterministic and nondeterministic programs. Theoret. Comput. Sci. 43, 123-147 (1986).

[3] Berghammer R., Schmidt G.: The RELVIEW-system. In: Choffrut C., Jantzen M. (eds.): Proc. 8th Annual Symposium on Theoretical Aspects of Computer Science (STACS '91), Hamburg, Febr. 1991, LNCS 480, Springer, 535-536 (1991)

[4] Berghammer R.: Computing the cut completion of a partially ordered set – An example for the use of the RELVIEW-system. Report Nr. 9205, Fakultät für Informatik, Universität der Bundeswehr München (1992)

[5] Berghammer R., Gritzner T., Schmidt G.: Prototyping relational specifications using higher-order objects. In: Heering, J., Meinke, K., Möller, B., Nipkow, T. (eds.): Proc. Int. Workshop on Higher Order Algebra, Logic and Term Rewriting (HOA 93), Amsterdam, The Netherlands, Sept. 1993, LNCS 816, Springer, 56-75 (1994) Extended version available as Report Nr. 9304, Fakultät für Informatik, Universität der Bundeswehr München (1993)

[6] Berghammer R., von Karger B., Ulke C.: Relation-algebraic analysis of Petri nets with RELVIEW. In: Margaria T., Steffen B. (eds.): Proc. 2nd Workshop on Tools and Applications for the Construction and Analysis of Systems (TACAS '96), Passau, March 1996, LNCS 1055, Springer, 49-69 (1996)

[7] Berghammer R., von Karger B.: Algorithms from relational specification. In: Brink C., Kahl W., Schmidt G. (eds.): Relational methods in Computer Science, Advances in Computing Science, Springer, 131-149 (1997)

[8] Chin L.H., Tarski A.: Distributive and modular laws in the arithmetic of relation algebras. University of California Publications in Mathematics (new series) 1, 341-384 (1951).

[9] Dijkstra E.W.: Hierarchical ordering of sequential processes. Acta Informatica 2, 143-161 (1973)

[10] Dijkstra E.W.: A discipline of programming. Prentice Hall (1976)

[11] Erné M.: Einführung in die Ordnungstheorie. B.I.-Wissenschaftsverlag (1982)

[12] Fraenkel A.S.: Planar kernel and Grundy with $d \leq 3, d_{out} \leq 2, d_{in} \leq 2$ are NP-complete. Disc. Appl. Math. 3, 257–262 (1981)

[13] Hattensperger C., Berghammer R., Schmidt G.: RALF – A relation-algebraic formula manipulation system and proof checker. In: Nivat M., Rattray C., Rus T., Scollo G. (eds.): Proc. 3rd Conference on Algebraic Methodology and Software Technology (AMAST '93), University of Twente, Niederlande, June 1993, Workshops in Computing, Springer, 407-408 (1993)

[14] Hermes H.: Einführung in die Verbandstheorie. 2. Auflage, Springer (1967)

[15] Hoffmann T.: Formale Verifikation relationaler Programme mittels Relationenalgebra und wp-Kalkül. Diploma Thesis, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, (1997)

[16] Jónsson B., Tarski A.: Boolean algebras with operators, Part II. Amer. J. Math. 74, 127-167 (1952).

[17] Lautenbach K.: Liveness in Petri nets. Bericht 02.1/75-7-29, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin (1975)

[18] Mathematics of Program Construction Group Eindhoven: Fixed-point Calculus. Information Processing Letters 53, 131–136 (1995)

[19] Reisig W.: Petri nets – An introduction. EATCS Monographs on Theoret. Comput. Sci., Springer (1985)

[20] Schmidt G., Ströhlein T.: On kernels of graphs and solutions of games: A synopsis based on relations and fixed points. SIAM J. Alg. Disc. Meth. 6,1, 54–65 (1985)

[21] Schmidt G., Ströhlein T.: Relationen und Graphen. Springer (1989); English version: Relations and graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoret. Comput. Sci., Springer (1993)

[22] Tarski A.: On the calculus of relations. Journal of Symbolic Logic 6, 73-89 (1941).

[23] Tarski A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5, 285 -309 (1955)

[24] Ulke C.: Rechnergestützte Spezifikation und Entwicklung relationaler Algorithmen. Diploma Thesis, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik (1995)

[25] Warshall S.: A theorem on boolean matrices. J. Assoc. Comput. Mach. 9, 11-12 (1062)

[26] Winter M.: Program verification with RELVIEW. In: Buth B., Berghammer R. (eds.): Systems for computer-aided specification, development and verification. Proc. Workshop 'Programmsysteme für rechnergestützte Programmentwicklung und -verifikation', Kiel, July 1994, Bericht Nr. 9416, Institut für Informatik und Praktische Mathematik, Universität Kiel, 79-90 (1994)

[27] Wolf A.: Relationale Behandlung von Inzidenzgraphen. Diploma Thesis, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik (to appear 1997)

[28] Zierer H.: Relation algebraic domain constructions. Theoret. Comput. Sci. 87, 163–188 (1991)

A Configuration of RELVIEW – Resources

As described in detail in Section 3.19.1, the RELVIEW system can be configured by setting resources in a configuration file. If a resource is not explicitly set, the system uses a default value as shown in the list below. The following resources are supported by RELVIEW:

Resource	example	(default)
# Size and position of		
# the graph editor window:		
*relview.graph.xv_width:	300	(400)
*relview.graph.xv_height:	300	(400)
*relview.graph.xv_x:	760	(10)
*relview.graph.xv_y:	490	(10)
# Size and position of		
# the relation editor window:		
*relview.relation.xv_width:	300	(400)
*relview.relation.xv_height:	300	(400)
*relview.relation.xv_x:	760	(10)
*relview.relation.xv_y:	160	(10)
# Size and position of		
# the menu-window:		
*relview.xv_width:	245	(340)
*relview.xv_height:	220	(750)
*relview.xv_x:	760	(10)
*relview.xv_y:	0	(10)
# Size and position of		
# the directory window:		
*relview.dir.xv_width:	330	(450)
*relview.dir.xv_height:	600	(950)
*relview.dir.xv_x:	600	(10)
*relview.dir.xv_y:	10	(10)
# Number of lines in the various		
# scroll lists in the directory window:		
Rel_dir.Lines:	5	(10)
Fun_dir.Lines:	5	(10)
Prog_dir.Lines:	5	(10)
Dom_dir.Lines:	2	(3)
# Specific button-resources:		
# button-width (panel_label_width)		
# A value of 0 means that the width of the		

```
# button depends on the width of the text
# in the button.
# xv_x and xv_y define the position of the
# button relatively to the top left corner
# of the window.

# Buttons of the menu window:
*relview*.files_button.panel_label_width:     0      (40)
*relview*.files_button.xv_x:                  8      (8)
*relview*.files_button.xv_y:                  4      (4)
*relview*.info_button.panel_label_width:      0      (40)
*relview*.info_button.xv_x:                   130    (176)
*relview*.info_button.xv_y:                   4      (4)
*relview*.quit_button.panel_label_width:      0      (40)
*relview*.quit_button.xv_x:                   190    (260)
*relview*.quit_button.xv_y:                   4      (4)

*relview*.rel_button.panel_label_width:       0      (40)
*relview*.rel_button.xv_x:                    8      (8)
*relview*.rel_button.xv_y:                    68     (68)
*relview*.graph_button.panel_label_width:     0      (40)

*relview*.xrv_button.panel_label_width:       0      (40)
*relview*.xrv_button.xv_x:                    8      (8)
*relview*.xrv_button.xv_y:                    132    (132)
*relview*.label_button.panel_label_width:     0      (40)

*relview*.fun_button.panel_label_width:       0      (40)
*relview*.fun_button.xv_x:                    8      (8)
*relview*.fun_button.xv_y:                    196    (196)
*relview*.eval_button.panel_label_width:      0      (40)
*relview*.iter_button.panel_label_width:      0      (40)
*relview*.test_button.panel_label_width:      0      (40)
*relview*.test_button.xv_x:                   8      (8)
*relview*.test_button.xv_y:                   228    (228)

*relview*.or_button.panel_label_width:        0      (40)
*relview*.or_button.xv_x:                     8      (8)
*relview*.or_button.xv_y:                     292    (292)
*relview*.and_button.panel_label_width:       0      (40)
*relview*.neg_button.panel_label_width:       0      (40)
*relview*.komp_button.panel_label_width:      0      (40)
*relview*.trans_button.panel_label_width:     0      (40)

*relview*.lres_button.panel_label_width:      0      (40)
```

relview.lres_button.xv_x:	8	(8)
relview.lres_button.xv_y:	356	(356)
relview.rres_button.panel_label_width:	0	(40)
relview.syq_button.panel_label_width:	0	(40)

relview.transc_button.panel_label_width:	0	(40)
relview.transc_button.xv_x:	8	(8)
relview.transc_button.xv_y:	420	(420)
relview.reflc_button.panel_label_width:	0	(40)
relview.symmc_button.panel_label_width:	0	(40)

relview.dom_def_button.panel_label_width:	0	(40)
relview.dom_def_button.xv_x:	8	(8)
relview.dom_def_button.xv_y:	484	(484)
relview.dom_ord_button.panel_label_width:	30	(40)
relview.1st_button.panel_label_width:	30	(40)
relview.2nd_button.panel_label_width:	30	(40)

relview.p1_button.panel_label_width:	30	(40)
relview.p1_button.xv_x:	8	(8)
relview.p1_button.xv_y:	548	(548)
relview.p2_button.panel_label_width:	30	(40)
relview.ptup_button.panel_label_width:	30	(40)

relview.s1_button.panel_label_width:	30	(40)
relview.s1_button.xv_x:	8	(8)
relview.s1_button.xv_y:	612	(612)
relview.s2_button.panel_label_width:	30	(40)
relview.stup_button.panel_label_width:	30	(40)

relview.epsi_button.panel_label_width:	0	(40)
relview.epsi_button.xv_x:	8	(8)
relview.epsi_button.xv_y:	676	(676)
relview.partf_button.panel_label_width:	0	(40)
relview.totf_button.panel_label_width:	0	(40)
relview.inj_button.panel_label_width:	0	(40)

Resources for the file-chooser:
position and size of the window

*file_chooser.xv_x:	10	(10)
*file_chooser.xv_y:	10	(10)
*file_chooser.xv_width:	580	(580)
*file_chooser.xv_height:	300	(300)

Positions of the scroll lists:

```
*dir_list.xv_x:                        10          (10)
*dir_list.xv_y:                        10          (10)

*file_list.xv_x:                       250         (250)
*file_list.xv_y:                       10          (10)

# Positions of the input fields:
*name_field.xv_x:                      10          (10)
*name_field.xv_y:                      240         (10)

*filter_field.xv_x:                    10          (10)
*filter_field.xv_y:                    270         (270)

# Button width and position:
*load_button.panel_label_width:        50          (50)
*load_button.xv_x:                     500         (500)
*load_button.xv_y:                     20          (20)

# Label of load button:
Load.Label:                            LOAD *.*    (LOAD *.*)

*save_button.panel_label_width:        50          (50)
*save_button.xv_x:                     500         (500)
*save_button.xv_y:                     50          (500)

Save.Label:                            SAVE *.xrv  (SAVE *.xrv)

*cancel_button.panel_label_width:      0           (50)
*cancel_button.xv_x:                   500         (500)
*cancel_button.xv_y:                   100         (100)

Cancel.Label:                          CANCEL      (CANCEL)

# Buttons for predefined filter values:
# (button width and position)
*xrv_button.panel_label_width:         60          (50)
*xrv_button.xv_x:                      500         (500)
*xrv_button.xv_y:                      150         (150)
*prog_button.panel_label_width:        60          (50)
*prog_button.xv_x:                     500         (500)
*prog_button.xv_y:                     190         (190)
*label_button.panel_label_width:       60          (50)
*label_button.xv_x:                    500         (500)
*label_button.xv_y:                    230         (230)
```

# Labels of filter buttons:		
Xrv.Label:	*.xrv	(*.xrv)
Prog.Label:	*.prog	(*.prog)
Label.Label:	*.label	(*.label)

B Example of a Start-up File

This section shows a number of functions which normally are stored in a file "start_up.xrv" resp. ".start_up.xrv" and loaded into the system at startup time. Startup files are described in detail in Section 3.19.2.

Name and parameters	Relational term	Meaning
`max(X,Y)`	`min(Y^,X)`	Maximal elements of a set wrt. an order
`min(X,Y)`	`Y&((X&-I(X))\-Y)`	Minimal elements of a set wrt. an order
`ma(X,Y)`	`mi(X^,Y)`	Upper bounds of a set wrt. an order
`mi(X,Y)`	`X/Y^`	Lower bounds of a set wrt. an order
`sup(X,Y)`	`inf(X^,Y)`	Least upper bound of a set wrt. an order
`inf(X,Y)`	`ge(X,mi(X,Y))`	Greatest lower bound of a set wrt. an order
`ge(X,Y)`	`le(X^,Y)`	Greatest element of a set wrt. an order
`le(X,Y)`	`Y&mi(X,Y)`	Least element of a set wrt. an order
`tc(X)`	`trans(X)`	Transitive closure
`rtc(X)`	`refl(trans(X))`	Reflexive-transitive closure
`sc(X)`	`X|X^`	Symmetric closure
`rc(X)`	`refl(X)`	Reflexive closure
`aec(X)`	`rtc(sc(X))`	Equivalence closure