A Graphical Interface Formalism: Specifying Nested Relational Databases

by

Geert-Jan Houben and Jan Paradaens

88/11

June, 1988
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.
A Graphical Interface Formalism: Specifying Nested Relational Databases

Geert-Jan Houben
Eindhoven University of Technology

Jan Paredaens
University of Antwerp

June 9, 1988

Abstract

An interface is considered as an automaton, for which the dynamics are represented by transitions on a set of states. Part of the actual state is currently represented by the screen. As such a program in $R^1$ represents the possible dialogue between the user and the system. An overview of $R^1$, a language for specifying interfaces, is given. $R^1$ is illustrated by specifying the $R^2$-interface, which is a graphical interface for handling nested relational data. The definition of the $R^2$-interface is such that queries on a database can be expressed in a way that suits the nested relational model, which means that objects at arbitrary levels can be specified directly, i.e., without referring to objects at other levels.

Keywords: graphical interfaces; menus; icons; nested relational data; query languages; complex objects.

1 Introduction

In current database research there is a trend towards easier use of database systems. In the field of relational databases there has recently been a lot of attention to nested structures [PA,RKS,S,SPS,SS,TF]. From this the Nested Relational Database Model (NRDM) has evolved, in which the constraint that databases have to be in First- Normal-Form has been relaxed. The basic difference between the classical Relational Database Model (RDM) and the NRDM is that, whereas in both models relations are sets of tuples, in the RDM the components of tuples have atomic values and in the NRDM the components may have structured values, which means that they may be relations themselves.

For the flat relations from the RDM there are a number of well-known formalisms for expressing queries. The move towards nested structures has led to the introduction of new formalisms suitable for expressing queries in a nested structure. An example of such a formalism is the nested algebra [GVG1,GVG2,OOM,PVG,VG], which is an extension of the relational algebra. In [HPT1,HPT2] the authors studied the expressibility of the nested algebra. The Verso model [AB] has many of the features essential to handle nested relations. This model is a good step in the direction of the model of complex objects. In [HP] a model was defined with a query formalism, the $R^2$-algebra, that not only has many of the features that make the
other models useful. It also has operations to handle computable information with operations like aggregation and computation, and with a more general notion of selection. Also recursion is embedded in the model.

The main characteristic of the $R^2$-algebra however is the fact that the query formalism has been designed having in mind that queries on the database should be specified through a graphical interface. Recently, graphical query languages and graphical interfaces [K,KK,KKS] are often suggested as an easy-to-use means of communicating with a database system.

The big advantage of a graphical interface is the intuitive way of specifying queries. This implies that systems offering such interfaces are easier-to-use by inexperienced users. Without a lot of training they are able to express queries themselves, since they do not need to learn languages that, because of their mathematical origin, are difficult to use for the average database user.

Using the advantages of the recent graphical techniques, like multiple windows, icons, pointing devices and several kinds of menus, operations can be specified directly without referring explicitly to the access path or the environment of the required values.

The $R^2$-algebra is designed with a graphical interface in mind. It is based on the nested algebra, which means that we want to define operations that have at least the power of the nested algebra. Besides giving the possibility of specifying operations at deeply nested levels in a direct way, this interface also has operations to compute aggregates and to apply sequences of operations recursively. Further there is an extension such that the functions used within operations like selection and aggregation (in order to compute the selection condition or the aggregation result) can be chosen not only from a set (menu) of standard functions, but also from a set of functions specified by the user himself. These latter functions are specified to the system by entering a program that the system is able to run whenever the function has to be applied.

Since operations in the $R^2$-algebra mainly are manipulations of schemes and since schemes are represented in the interface by trees, the operations in the interface are specified by means of clicking nodes in trees and of choosing items from menus. This implies that in order to express queries the user must manipulate trees such that trees are obtained that represent the resulting relations.

In this paper we use the language $R^1$ to define which manipulations in the interface specify which operations in the $R^2$-algebra. In $R^1$ it is described for each operation which menu options have to be chosen and which nodes have to be clicked, and how this affects the system's state. The system's state contains information about the relations known to the system and about which of them are represented on the screen.

The language $R^1$ can also be used to specify other interfaces. The strategy that
we use with $R^1$ can easily be used for specifying other graphical interfaces. The big advantage of this language is the ease with which the usage of an interface can be described. Usually the formal definition of the system's operations is hard to understand. The task of the interface is to abstract from these difficult operations. This also implies that the definition of the use of the interface must be easy to understand.

Another advantage of the interface specification is the independence from the implementation of (in our case) the query formalism. In the specification of the $R^2$-interface we use very often strings. After we have defined how the two-dimensional figures (trees) are translated in (one-dimensional) strings, the manipulations of the trees are defined by the manipulations of the corresponding trees. This implies that we can easily choose other two-dimensional figures as representation of the relations without bothering about the specification of the operations.

As will become clear during the presentation of the $R^2$-algebra this algebra is likely to be extended to cover items like complex objects. Whenever attributes start living their own live, using user-defined operations (as with the selection and the aggregation), then they can be viewed as representations of complex objects. The hierarchical approach used within the NRDM is just a good point to start from. The specification of an interface for such a model for complex objects can be very much the same as for the $R^2$-interface, when $R^1$ is used to specify the interface.

In this paper the language $R^1$ for specifying graphical interfaces is described. The $R^2$-interface is described in section 2. In section 3 we specify in $R^1$ the basic operations of the $R^2$-interface. The aggregation and the computation are covered in section 4, whereas the definition of programs and the (recursive) execution of programs are subject of section 5.

2 $R^2$-interface

The purpose of this paper is to demonstrate $R^1$ by describing the $R^2$-interface in $R^1$. This interface is defined in [HP].

The basic concepts of the $R^2$-algebra are analogous to those used in the Nested Algebra (NA). Both are related to the NRDM, where relations are viewed as sets of tuples that have as components either atomic data values or sets of tuples. The structure of a relation is defined by its scheme. A scheme is a list of attributes, where attributes are associated with a set of atomic data values (domain) or they are schemes themselves. Just like in the NA the operations in the $R^2$-algebra mainly use structural information, i.e. they mainly depend on the scheme. This is also the reason why we choose to represent relations by the representation of their scheme and to hold the corresponding instance in the background.

Now we will consider the concepts of the $R^2$-algebra. and their representation in $R^1$, in more detail.
A scheme of a relation is an identifier followed by a list of attributes enclosed within brackets, where an attribute is either atomic or structured. An atomic attribute is an identifier, called the name of that attribute. A structured attribute is a scheme, where the name of that scheme is called the name of the structured attribute. All identifiers (names) in a scheme must be different.

With every atomic attribute \( \alpha \) a set, called its domain, \( \text{dom}(\alpha) \) is associated, for instance the set of natural numbers. If \( \eta(\alpha_1, ..., \alpha_n) \) is a scheme \( \lambda \) with name \( \eta \) where \( \alpha_i \) is an attribute, then the domain of this scheme \( \text{dom}(\lambda) \) is equal to the Cartesian product of the attributes \( \text{dom}(\alpha_1) \times .. \times \text{dom}(\alpha_n) \). The set of instances of scheme \( \lambda \), denoted by \( \text{Inst}(\lambda) \), is the set of finite subsets of \( \text{dom}(\lambda) \). The elements of an instance of scheme \( \lambda \) are called tuples over \( \lambda \).

An example of a scheme with name \textit{students} and with four attributes, two of which are structured (\textit{addresses} and \textit{exams}), is

\[
\textit{students}(\text{name}, \text{addresses}(\text{street}, \text{nr}, \text{city}), \text{year}, \text{exams}(\text{subject}, \text{date}, \text{result})).
\]

In the interface a scheme is represented by a tree, for which the nodes correspond to the attributes of the scheme. A directed edge of the tree from node \( x \) to node \( y \) denotes that the attribute corresponding to \( x \) is structured and that the attribute corresponding to \( y \) is an attribute of its list. A node will be labeled with the name of the corresponding attribute.

The above mentioned scheme would be represented like:

![Diagram of the students scheme](image)

The representation of an instance could be arranged in the same tree-like manner. However, since the operations of the algebra mainly concern the schemes, we will not bother about the instances in this paper. An additional operation in the interface will be used to get the instance corresponding to a tree.

A possible \( R^2 \)-representation of an instance of the above scheme could be:
In $R^1$, the language to specify the interface, we choose to represent the above scheme by its formal algebraic representation:

$$\text{students(name, addresses(street, nr, city), year, exams(subject, date, result)).}$$

The above instance is expressed in $R^1$ by:

$$<\text{students} = \{<\text{name} = \text{John}, \text{addresses} = \{<\text{street} = \text{Road}, \text{nr} = 10, \text{city} = \text{Town'}>, <\text{street} = \text{Avenue}, \text{nr} = 7, \text{city} = \text{Village'}>\}, \text{year} = 1, \text{exams} = \{<\text{subject} = \text{Math}, \text{date} = 050288, \text{result} = 8'>\}>, <\text{name} = \text{Jim}, \text{addresses} = \{<\text{street} = \text{Street}, \text{nr} = 66, \text{city} = \text{City'}>\}, \text{year} = 0, \text{exams} = 0'>}.$$  

The main advantage of the use of trees is the two-dimensional (graphical) nature of the objects that have to be manipulated. Attributes (at any nested level) can be specified in a direct way, which means that not, as in the NA, a path from the root has to specified. This makes it possible to reason about nested attributes, partial objects, rather independent of their environment, which makes the reasoning more intuitive.

In our model we have chosen to represent a database as a set of relations. We
could have modeled a database as one nested relation (universal nested relation). By choosing for a set of relations we leave the user the freedom to specify what he wants. An additional advantage is the ease of defining different defaults, such as the representation of values, the operations allowed on the values and the user-defined functions and operations.

When formulating queries the starting point is a set of relations represented by a screen of trees (plus corresponding instances in background). From the menu of available operations the user chooses a sequence of operations in order to obtain a screen of trees that represent relations that contain the answer of the query. Every operation requires a number of arguments, among which the relation(s) on which the operation should be applied. This means that after choosing an operation, depending on the arity of the operation, one or two trees have to be pointed out. Depending on the operation, nodes in such a tree have to be clicked in order to specify attributes that play a role in the operation.

As an example we consider the query on the above relation where we want for every student his name and his addresses. Starting with the *students* tree on the screen the user first chooses the projection option from the menu containing all the possible operations. Then he clicks the *name* and *addresses* nodes and the system computes the new instance and shows the corresponding tree on the screen. After the user has entered a name for the new relation, *result* say, the relation with that name is represented on the screen and its instance contains the answer to the user's query.

3 Basic $R^2$-operations

In [HP] the formal definition of the operators of the $R^2$-algebra is given. Here we will present the definition of the interface for these operations, i.e. we define in $R^1$ the manipulations that are needed to specify an $R^2$-operation.

In order to specify manipulations in the interface we have three kinds of expressions in $R^1$.

- The first kind of manipulation specifies the picking of an item from some menu. The $R^1$ expression $[Menu \triangleright Item]$ denotes the choice of the item $Item$ from the menu $Menu$. For exactly defining the expressions in $R^1$ we use a global system state in which also the contents of the several menus are held.

- In order to denote the clicking of a node in a tree visible on the screen, we use $[Tree!Node]$. This expression will mean that in the tree with name $Tree$, the name of a tree being the label of the root, the node labeled $Node$ is clicked.

- The third kind of manipulation is the entering of a new label for a node in a tree computed by the system, but for which the label is not yet specified. For this we use the expression $[Tree; Node?Label]$, where $Node$ denotes the node of the tree $Tree$ that will get $Label$ as its label.
For defining the semantics of the operations we use a global system state in which we hold all the information that is important for the interface. MENU will be a function that assigns to each menu name the set of items that can be chosen from that menu. The domain of MENU will contain for example Operations and Relations. In MENU(Operations) we will have items like Union, Selection and Projection. RELATION will be a function that assigns to each scheme the instance corresponding to that scheme. The domain of RELATION will contain all those schemes that are known to the system and which can be shown on the screen. The range of RELATION holds all the instances that are held in the memory, but which can be represented on the screen. SCREEN is a subset of the domain of RELATION, which holds those schemes that are currently shown on the screen.

In this section we will specify in $R^1$ the basic $R^2$ operations, by which we mean those operations that come from the nested algebra: union, difference, join, projection, selection, renaming, nest and unnest.

First the binary operations which are the union, the difference and the join. The definition of these operators is a straightforward generalization of the nested algebra [HPT1], which implies that the operation can be applied at any level, not just at the first level.

In order to specify such a binary operation (on the screen) the user first has to specify which kind of operation he is interested in. This implies that in the Operations menu the operation is chosen. After the system knows that the operation is a union, say, the user has to specify the arguments. For specifying an entire relation the user has to click the root of the tree representing the relation. If the user wants to specify some structured attribute as argument of the operation then the node representing that attribute has to be clicked. Note that the order in which these two arguments are specified is important. The system now knows that a new relation has to be computed, which is either the union of two given relations or a relation with a new attribute that is the union of two given attributes. Therefore it computes the scheme and the instance of this new relation and it represents the scheme on the screen, i.e. a new tree appears. Note that at this stage the scheme is not yet complete since the unique name for the relation (the root of the tree) is not yet specified. This implies that the user's last activity is to enter that name. If the union is nested, i.e. a new attribute is constructed, then the name for that new attribute has to be specified also.

We will now specify a union in $R^1$, where conditions on the values of MENU, RELATION and SCREEN are stated between /* and */.

Consider two relations $r_1$ and $r_2$ with scheme $n_1(l)$ and $n_2(l)$ resp. and with instance $v_1$ and $v_2$ resp. Suppose we want to express $UNI[n_1(l);n_2(l);n_3](r_1,r_2)$, which denotes the union such that the result has scheme $n_3(l)$ and value $v_3$ (equal to $v_1 \cup v_2$).
For this union it is required that both the argument relations are represented at
the screen at the start of the operation. The result of the first three manipulations (;
denotes composition in $R^1$) is that $\text{dom} (\text{RELATION})$ and $\text{SCREEN}$ are augmented
with $\Box (l)$ and that $\text{RELATION}(\Box (l))$ equals the instance of the union of the relations
with schemes $n_1(l)$ and $n_2(l)$. We use $\Box$ as a special label for a node for which a new
name has to be specified. The fourth manipulation replaces in the scheme with name
$\Box$ the label $\Box$ by $n_3$, such that $n_3(l)$ is the scheme of $UNI[n_1(l); n_2(l); n_3]$, and
that $\text{RELATION}(n_3(l))$ is its instance.

Note that in the above specification the conditions on $\text{MENU}, \text{RELATION}$ and
$\text{SCREEN}$ that hold at the start are supposed to hold as long as they do not con­
tradict the new conditions. For instance, $n_1(l)$ stays in $\text{SCREEN}$ during the entire
operation.

In [HP] the exact definition of the binary operations at a nested level is not given,
but with [HPT2] it is easy to define $UNI[n(l); n_1; n_2; n_3; n']$ to denote the nested
union of the attributes $n_1$ and $n_2$ within scheme $n(l)$, such that a new relation is
computed with scheme $n'(l')$, where in $l'$ $n_3$ is the new attribute that contains the
union of $n_1$ and $n_2$.

Suppose $r$ is a relation with scheme $n(l)$ and instance $v$, where $r'$ equal to
$UNI[n(l); n_1; n_2; n_3; n']$ is to be computed. Let the scheme of $r'$ be $n'(l')$ and its
instance $v'$. Suppose $l''$ is $l'$ with $n_3$ replaced by $\Box$. Then the specification of the
union is defined by :

```c
/* Union ∈ MENU(Operations) */
/* {n_1(l), n_2(l)} ∈ SCREEN */
/* RELATION(n_1(l)) = v_1, RELATION(n_2(l)) = v_2 */
[ Operations ▸ Union ]; [n_1!n_1]; [n_2!n_2];
/* □(l) ∈ SCREEN, RELATION(□(l)) = v_3 */
[□; □?n_3]
/* □(l) ∉ dom(RELATION) */
/* n_3(l) ∈ SCREEN, RELATION(n_3(l)) = v_3 */
```
nipulation nodes in the same tree are clicked, the nested union is computed and a new name for the new attribute is required.

As an illustration of the nested union we consider in the next figure the application of the union

\[ UNI[parents(sons(child), daughters(child)); sons; daughters; children; persons] \]

on a relation represented by the tree with name parents. Its result would be a relation represented by the tree with name persons.

![Diagram of nested union](image)

The specification for the difference and the join are basically the same, however for the join the \( R^2 \)-algebra specifies other conditions for the schemes of the arguments, i.e. they may not contain attributes with the same name.

For the projection a similar strategy is followed. A projection is such that one specifies a number of nodes in the tree of a relation, where the nodes represent the attributes that one wants to have in the scheme of the new relation. These nodes need not to be all at the first level, as in the nested algebra. However there is a constraint on which nodes can be specified [HP]. It is not allowed to project out nodes without projecting out all their descendants.

The definition can be such that the number of nodes to be specified is minimized. Here we will use, as in [HP], the rule that for every node clicked the corresponding attribute, its predecessor and its descendants are specified to occur in the new scheme.

Since the number of nodes to specify is not fixed the end of the specification has to be specified (using the EOL option in the Specification menu, say).

Let us consider the projection \( PRO[n(I); lan; n'] \) on the relation \( r \) with scheme \( n(I) \) and instance \( v \), where the result is \( r' \) with scheme \( n'(l') \) and instance \( v' \). Suppose \( lan = n_1, \ldots, n_k \).

/* Projection \( \in \) MENU(Operations), EOL \( \in \) MENU(Specification) */
/* \( n(I) \) \( \in \) SCREEN, RELATION(\( n(I) \)) = v */
[ Operations \( \Rightarrow \) Projection ];
As an illustration of a projection we consider applying

\[ \text{PRO[studs - scheme; name, city; citizens]} \]

on the \textit{students} relation of section 2, where we use \textit{studs - scheme} as a shorthand for the scheme of \textit{students}. The next figure shows the tree of the resulting relation.

\[
\begin{array}{c}
\text{citizens} \\
\text{name} \quad \text{addresses} \\
\quad \text{city}
\end{array}
\]

The main characteristic of the selection results from the possibility in the \(R_2\)-algebra of choosing the selection function from a set of user-defined functions. This implies a menu of selection functions. For every item in that menu the system has an algorithm that, given some attributes for the function’s parameters, serves as a selection criterion.

Let us consider the selection \(SEL[n(l); f; lan; n']\) on the relation \(r\) with scheme \(n(l)\) and instance \(v\), where the result \(r'\) has scheme \(n'(l')\) and instance \(v'\). Suppose \(lan = n_1, \ldots, n_k\) and \(f\) a function with \(k\) parameters, such that with \(n_1, \ldots, n_k\) for these parameters a selection function is specified.

\[
\begin{array}{c}
/* \text{Selection } \in \text{MENU(Operations)}; f \in \text{MENU(Selection Functions)} */ \\
/* n(l) \in \text{SCREEN, RELATION}(n(l)) = v */ \\
[\text{Operations } \triangleright \text{Selection ];} \\
[\text{Selection Functions } \triangleright f ];[n!n_1]; \ldots;[n!n_k]; \\
/* \Box(l) \in \text{SCREEN, RELATION}(\Box(l)) = v' */ \\
[\Box; \Box?!n'] \\
/* \Box(l) \notin \text{dom(RELATION)} */ \\
/* n'(l') \in \text{SCREEN, RELATION}(n'(l')) = v' */
\end{array}
\]

Note that we suppose that the system is able to deduce from the selection function the number of nodes to be clicked, so that it is not required to choose EOL to specify the end of the list.

If \(f\) is a function that checks whether a value is equal to 1, then the application of \(SEL[studs - scheme; f; year; first - year - students]\) would result in a relation
with exactly the same tree as the students tree, except for the root which is now labeled first - year - students.

An other operation is the renaming. It is used to substitute given attribute names by new names. This is achieved by clicking some nodes and by then entering new names for those nodes. We choose to do this for several nodes at a time.

Consider $REN[n(l); lan; lan'; n']$ on the relation $r$ with scheme $n(l)$ and instance $v$, resulting in $r'$ with scheme $n'(l')$, where $l'$ equals $l$ with the attributes from $lan = n_1, \ldots, n_k$ replaced by the corresponding ones from $lan' = n'_1, \ldots, n'_k$.

\[
\begin{align*}
&\text{Consider } REN[n(l); lan; lan'; n'] \text{ on the relation } r \text{ with scheme } n(l) \text{ and instance } v, \\
&\text{resulting in } r' \text{ with scheme } n'(l'), \text{ where } l' \text{ equals } l \text{ with the attributes from } lan = n_1, \ldots, n_k \text{ replaced by the corresponding ones from } lan' = n'_1, \ldots, n'_k.
\end{align*}
\]

Two operations typical for the nested algebra are the nest and the unnest. With the nest it is possible to construct, given a number of attributes from one list of attributes, a new structured attribute with those attributes as its attributes.

Consider the nest $NES[n(l); lan; lan'; n']$ on relation $r$ with scheme $n(l)$ and instance $v$. The result $r'$ will have scheme $n'(l')$ and instance $v'$. Suppose $lan = n_1, \ldots, n_k$ and $l'$ is $l$ with $lan$ (the new attribute's name) replaced by $\sqcup_2$.

\[
\begin{align*}
&\text{Consider the nest } NES[n(l); lan; lan'; n'] \text{ on relation } r \text{ with scheme } n(l) \text{ and instance } v, \\
&\text{resulting in } r' \text{ with scheme } n'(l') \text{ and instance } v'. \text{ Suppose } lan = n_1, \ldots, n_k \text{ and } l' \text{ is } l \text{ with } lan \text{ (the new attribute's name) replaced by } \sqcup_2.
\end{align*}
\]
Consider the application of

\[nes\{\textit{studs} - \textit{scheme}; \textit{year}, \textit{exams}; \textit{education}; \textit{persons}\}\]

on the \textit{students} relation. The resulting tree is shown in the next figure.

![Tree diagram](image)

Using the unnest it is possible to substitute for a structured attribute the list of its attributes. So, consider \textit{UNN}\{n(l); u; n\}.

\[\text{Unnest} \in \text{MENU(Operations)}\]

\[\text{* } n(l) \in \text{SCREEN, RELATION}(n(l)) = v^* /\]

\[\text{Operations } \triangleright \text{ Unnest } ; \{\text{ntu}];\]

\[\text{* } \Box_t(l') \in \text{SCREEN, RELATION}(\Box_t(l')) = v^* /\]

\[\text{Operations } \triangleright \text{ Unnest } ; \{\text{ntu}];\]

\[\text{* } \Box_t(l') \notin \text{dom(RELATION)} /\]

\[\text{* } n'(l') \in \text{SCREEN, RELATION}(n'(l')) = v^* /\]

The application of \textit{UNN}\{\textit{studs} - \textit{scheme}; \textit{addresses}; \textit{persons}\} would lead to a
relation with the next tree.

4 Aggregation and Computation

Two operations in the $R^2$-algebra that do not originate from the nested algebra are the aggregation and the computation. These are operations that do not handle structural information as in the nested algebra, but they concern computable information [HPT1]. The aggregation computes new values based on values of a structured attribute, whereas the new values of the computation are based on values of a tuple.

An aggregation is defined by giving a number of attributes from the list of some structured attribute $\alpha$. For every value of $\alpha$, which is a set of tuples, the multiset of subtuples over these attributes is computed. Using some aggregation function a new value is created from this multiset. Typical aggregation functions would be functions that compute the sum or the average of a set of numbers. The new values become values of a new attribute, for which the node is a sibling of that of $\alpha$.

Consider the aggregation $AGG[n(l); f; \lambda n; a{n}; n']$ on relation $r$ with scheme $n(l)$ and instance $v$. Let $f$ be an aggregation function, which can have the attributes from $\lambda n = n_1, \ldots, n_k$ as parameters and which produces values of the new attribute $a{n}$. The result $r'$ will have scheme $n'(l')$ and instance $v'$. Let $l''$ be $l'$ with $a{n}$ replaced by $\Box_2$.

/* Aggregation \in MENU(Operations) */
/* $f \in MENU(Aggregation \ Functions) */
/* $n(l) \in SCREEN, RELATION(n(l)) = v */
[ Operations $\triangleright$ Aggregation ];
[ Aggregation \ Functions $\triangleright$ f ];[\lambda n_1; \ldots; \lambda n_k];
/* $\Box_1(l'') \in SCREEN, RELATION(\Box_1(l'')) = v' */
[\Box_1;\Box_2?a{n}];
/* $\Box_1(l'') \notin dom(RELATION) */
/* $\Box_1(l') \in SCREEN, RELATION(\Box_1(l')) = v' */
[\Box_1;\Box_1?n'];
/* $\Box_1(l') \notin dom(RELATION) */
/* $n'(l') \in SCREEN, RELATION(n'(l')) = v' */
If \textit{sum} is a function that computes for a set of numbers the sum of those numbers, then the result of \textit{AGG[studs - scheme; sum; result; total; student - results]} would be a tree like:

![Tree Diagram]

As in the $R^2$-algebra we require that the parameter attributes of the aggregation and its result attribute are atomic. The system should be able to interpret a choice from the Aggregation Functions menu in such a way that a user-defined algorithm is used to compute new values with the specific attributes as parameters to the algorithm.

The computation operation is a variant of the aggregation where a new value is computed based on one tuple rather than on a set of tuples. This implies that the new attribute becomes a sibling attribute of the attributes, that are specified as the parameters of the computation function, instead of a sibling of their parent.

The specification is analogous to that of the aggregation using the Computation option in the Operations menu and using the Computation Functions menu.

5 Recursion

With the previously introduced operations the user is able to specify single algebraic operations. The interface must supply the user with the possibility to specify sequences of operations that can be applied recursively, thus specifying recursive expressions. This implies the ability to specify sequences of operations and to specify that sequences are applied recursively.

For the recursion a stop criterion is required. As in [HP] we define that a sequence of operations is executed again and again as long as the instances do not stay unchanged. In our interface this means that the recursion is stopped, whenever an execution of the sequence does not change SCREEN and RELATION anymore. We could have defined other stop criteria and a menu from which the desired criterion could be picked, but we will use only the above criterion.

Applying a sequence of operations recursively is a special case of applying a sequence of operations, called a program. In our interface a program is a specification of a sequence of operations. A program can have relations (i.e. schemes) as parameters in order to be able to apply programs to relations, that are different but that have schemes on which the same operations can be applied.

In order to define a program one must switch from normal mode to programming
mode (the Program option from the Specification menu). Subsequently operations are specified, just as described before. The system knows that it only has to deal with the schemes, since the instances play a role only at the time of the actual application. The program specification is ended by choosing EOL from Specification. The system then asks for a name of the program and stores the program such that if that name is chosen from the Programs menu the system knows what to do.

In [HP] it was defined that programs started with sets of relations and ended with sets of relations, thus being able to consider only relevant relations, not intermediate results. In the interface this is specified by the state of SCREEN at the beginning and at the end of the program definition, where Delete Tree is used (from the Specification menu) to erase trees from the screen. This implies during the definition phase that for the elements of SCREEN the RELATION value is not defined (i.e., they have the value $\bot$, say).

For the execution of a program (not recursively) the Execute Program option from Operations has to be chosen and then the desired program has to be specified in Programs. Note that the state of SCREEN must be such that the relations represented on the screen correspond to the schemes that were in SCREEN at the beginning of the definition of that program.

Now the system is able to compute new relations according to the program, thus altering SCREEN and RELATION. The new trees should then represent the relations in which the user was interested.

For the recursive execution the specification is the same, except for the choice of Recursive Program from Operations. With the recursion the state of SCREEN at the end of one execution of the program must be such that for another execution SCREEN has the proper starting value. Usually this requires some renaming of relations, i.e., replacing old trees by new trees with old names.

Let us consider the program $p$ that starts from two relations with schemes $n_1(l_1)$ and $n_2(l_2)$ and that produces a new relation with scheme $n_3(l_3)$, that is renamed to $n_2(l_2)$, whereas the original $n_2(l_2)$ is renamed to $n_1(l_1)$. Suppose $v_1$ and $v_2$ are the instances corresponding to $n_1(l_1)$ and $n_2(l_2)$ at the start and suppose $v_3$ corresponds to $n_3(l_3)$.

Executing this program once is specified as:

```plaintext
/* Execute Program ∈ MENU(Operations), p ∈ MENU(Programs) */
/* \{n_1(l_1), n_2(l_2)\} = SCREEN */
/* RELATION(n_1(l_1)) = v_1, RELATION(n_2(l_2)) = v_2 */
[ Operations ⊃ Execute Program ];
[ Programs ⊃ p ]
/* \{n_1(l_1), n_2(l_2)\} = SCREEN */
/* RELATION(n_1(l_1)) = v_2, RELATION(n_2(l_2)) = v_3 */
```

If we suppose that $p$ could be applied to two relations with the trees shown in
the next figure, then the application of \( p \) would not change anything on the screen, however the corresponding instances have probably changed.

![Diagram](image)

The only difference of recursive execution is the choice of Recursive Program. The resulting state of SCREEN and RELATION is such that another application of \( p \) would not change anything.

At the moment the interface does not supply the user with the possibility of applying programs to parts of relations. The idea of that generalization is that if a program requires a scheme with a specific structure, then the program can also be applied to relations with that scheme as subscheme. At instance level this implies that the system has to compute a resulting instance starting from the instance with every value of that subscheme considered separately.

6 Conclusions

The language \( R^1 \) for the specification of graphical interfaces is introduced. Any interface is considered as an automaton, for which the dynamics are represented by transitions on a set of states. Part of the actual state is currently represented by the screen. As such a program in \( R^1 \) represents the possible dialogue between the user and the system.

Using \( R^1 \) a graphical interface for nested relational databases is described. The organization of the information within the system is specified (MENU, RELATION, SCREEN). The usage of the interface is described by defining the semantics of several kinds of manipulations, like clicking on the screen, choosing from menus and entering data from the keyboard. All in all we have specified a uniform way of communicating with the database system. This uniform way of handling the data within the system makes it easy for non-experienced users to work with the system through this interface.

Many of the essential features for graphical interfaces of database systems [K] are integrated within the \( R^2 \)-interface. Queries are specified by the user in a piecemeal manner. With the information concerning the structure of the database always on the screen, the user formulates queries by constructing new relations using basic operations from the underlying formalism. In this way the user is able to express complex queries from the \( R^2 \)-algebra in an intuitive and uniform way.

The strategy that we have followed with \( R^1 \) was to first define one-dimensional representations for the two-dimensional figures that are shown on the screen and that give the user insight of the information currently held in the system. Then the
manipulations that the user is able to execute are defined by specifying their implications for the system's state. The basic purpose of $R^1$ is to specify the interface's functionality. An additional advantage of the use of the one-dimensional formalism is the possibility of telling in an unambiguous way what the interface must do without already designing the lay-out of screens and other non-functional features.

When specifying graphical interfaces (for database systems) the need for a language like $R^1$ is very obvious. In the design phase of such an interface one first of all wants to specify the functionality of the system. At that moment one does not want to bother very much about the figures on the screen but one wants to specify which information is represented on the screen. Furthermore it is important to specify which manipulations are possible and the effects that they cause.

Doing this in this uniform way makes it much easier for the designer to test the functionality by building prototypes and ask future user's to try to use this prototype for the work that they want to be able to do with the real system. Also specifications like these are much better to use when the actual implementation has to be made and this job has to be done by several people. With this formalism the specifications for the different tasks of the implementors can be stated in a more intuitive but precise way. Discussing an interface design is always easier using specification in a rather formal language. However the language is intuitive enough to be able to use it with future users that do not have any formal mathematical background.

Currently, one of the authors is engaged in a project to extend the database model used here to incorporate complex objects. The new model should not only allow for formulating queries but also for formulating updates in a uniform way. By viewing the nested attributes as complex objects, each with their own operations, the modeling power increases. Note that many existing implementations of object-oriented databases map the object-oriented systems to relational databases. Since however in the NRDM sets are a key issue, it is much cleaner to map complex objects to nested relational databases than to flat relations.

The big advantage is the increased embedding of dynamics within the database, thus being able to model the behaviour of real-world objects. One of the main items will be the integration of data manipulation and programming. This idea is in the $R^2$-interface already illustrated by the aggregation. When we have a language in which data definition, data manipulation and general computation are integrated in a uniform way, then the applications are much easier to design and to use. If we also have incorporated complex objects, then we are able to work at a higher level of abstraction, which also benefits a rapid design of database systems. Because of this approach prototyping will not cause the implementation to change completely.

For this model we can use an interface which is very similar to the $R^2$-interface. The specification in $R^1$ will therefore be very analogous to the one in this paper. An important extension will be that MENU will be structured in the same way as the information is structured. Every attribute will get its own menu's with possible operations. Whereas in the $R^2$-interface the selection functions were held centrally,
they can be made dependent on the attributes in the extended approach. Giving each attribute its own set of operations it is much easier to model the behaviour of complex real-world objects.

Also the introduction of operations for the modeling of updates will not cause any problems. We only have to add these operations to MENU and to make sure that the system knows how to interpret such operations. Using $R_1^1$ this can be solved in the same way as with the query formalism. One can define exactly how the system's state changes.

It must be clear that a language like $R_1^1$ can serve as a clean formalism for the specification of (graphical) interfaces. A system independent formalism like this can also help to make an integration between the system and its interfaces. During the design of the system a specification in $R_1^1$ of the necessary interfaces can help to guide the designers towards a completely integrated, though efficient system. Furthermore the design of an interface can benefit from a formalism like this because of the future user's ability to evaluate the specification (or the prototype) and to suggest improvements.

References


In this series appeared:

<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeff, H.M.J.L. Schols</td>
<td>Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing, K.M. van Hee, M. Voorhoeve</td>
<td>ELISA, A language for formal specifications of information systems</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben, J. Paredaens, K.M. van Hee</td>
<td>The partition of an information system in several parallel systems</td>
</tr>
<tr>
<td>86/05</td>
<td>Jan L.G. Dietz, Kees M. van Hee</td>
<td>A framework for the conceptual modeling of discrete dynamic systems</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeff</td>
<td>Nondeterminism and divergence created by concealment in CSP</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth, L. Shira</td>
<td>On proving communication closedness of distributed layers</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responder's commentary (IFIP86)</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher, R. Gerth</td>
<td>A timed failures model for extended communicating processes</td>
</tr>
<tr>
<td>86/14</td>
<td>R. Koymans</td>
<td>Specifying passing systems requires extending temporal logic</td>
</tr>
<tr>
<td>87/01</td>
<td>R. Gerth</td>
<td>On the existence of sound and complete axiomatizations of the monitor concept</td>
</tr>
<tr>
<td>87/02</td>
<td>Simon J. Klaver, Chris F.M. Verberne</td>
<td>Federatieve Databases</td>
</tr>
<tr>
<td>87/03</td>
<td>G.J. Houben, J. Paredaens</td>
<td>A formal approach to distributed information systems</td>
</tr>
<tr>
<td>87/04</td>
<td>T. Verhoeff</td>
<td>Delay-insensitive codes - An overview</td>
</tr>
<tr>
<td>87/05</td>
<td>R. Kuiper</td>
<td>Enforcing non-determinism via linear time temporal logic specification.</td>
</tr>
<tr>
<td>87/06</td>
<td>R. Koymans</td>
<td>Temporele logica specificatie van message passing en real-time systemen (in Dutch).</td>
</tr>
<tr>
<td>87/07</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems with real-time temporal logic.</td>
</tr>
<tr>
<td>87/08</td>
<td>H.M.J.L. Schols</td>
<td>The maximum number of states after projection.</td>
</tr>
<tr>
<td>87/10</td>
<td>T. Verhoeff</td>
<td>Three families of maximally nondeterministic automata.</td>
</tr>
<tr>
<td>87/11</td>
<td>P. Lemmens</td>
<td>Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.</td>
</tr>
<tr>
<td>87/12</td>
<td>K.M. van Hee, A. Lapinski</td>
<td>OR and AI approaches to decision support systems.</td>
</tr>
<tr>
<td>87/13</td>
<td>J.C.S.P. van der Woude</td>
<td>Playing with patterns, searching for strings.</td>
</tr>
<tr>
<td>87/14</td>
<td>J. Hooman</td>
<td>A compositional proof system for an occam-like real-time language</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>87/15</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>A compositional semantics for statecharts</td>
</tr>
<tr>
<td>87/16</td>
<td>H.M.M. ten Eikelder, J.C.F. Wilmont</td>
<td>Normal forms for a class of formulas</td>
</tr>
<tr>
<td>87/17</td>
<td>K.M. van Hee, G.-J.Houben, J.L.G. Dietz</td>
<td>Modelling of discrete dynamic systems framework and examples</td>
</tr>
<tr>
<td>87/18</td>
<td>C.W.A.M. van Overveld</td>
<td>An integer algorithm for rendering curved surfaces</td>
</tr>
<tr>
<td>87/19</td>
<td>A.J. Seebregts</td>
<td>Optimalisering van file allocatie in gedistribueerde database systemen</td>
</tr>
<tr>
<td>87/20</td>
<td>G.J. Houben, J. Paredaens</td>
<td>The $R^2$-Algebra: An extension of an algebra for nested relations</td>
</tr>
<tr>
<td>87/21</td>
<td>R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro</td>
<td>Fully abstract denotational semantics for concurrent PROLOG</td>
</tr>
<tr>
<td>88/01</td>
<td>T. Verhoeff</td>
<td>A Parallel Program That Generates the Möbius Sequence</td>
</tr>
<tr>
<td>88/02</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable Specification for Information Systems</td>
</tr>
<tr>
<td>88/03</td>
<td>T. Verhoeff</td>
<td>Settling a Question about Pythagorean Triples</td>
</tr>
<tr>
<td>88/04</td>
<td>G.J. Houben, J. Paredaens, D. Tahon</td>
<td>The Nested Relational Algebra: A Tool to handle Structured Information</td>
</tr>
<tr>
<td>88/05</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable Specifications for Information Systems</td>
</tr>
<tr>
<td>88/06</td>
<td>H.M.J.L. Schols</td>
<td>Notes on Delay-Insensitive Communication</td>
</tr>
<tr>
<td>88/07</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Modelling Statecharts behaviour in a fully abstract way</td>
</tr>
<tr>
<td>88/08</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>A Formal model for System Specification</td>
</tr>
<tr>
<td>88/09</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>A Tutorial for Data Modelling</td>
</tr>
<tr>
<td>88/10</td>
<td>J.C. Ebergen</td>
<td>A Formal Approach to Designing Delay Insensitive Circuits</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>88/11</td>
<td>G.J. Houben</td>
<td>A Graphical Interface Formalism: Specifying Nested Relational Databases</td>
</tr>
<tr>
<td></td>
<td>J. Paradaens</td>
<td></td>
</tr>
</tbody>
</table>