SID: a system for interactive design

by TOSIYASU L. KUNII and MINORU HARADA
The University of Tokyo
Hongo, Tokyo, Japan

INTRODUCTION

A System for Interactive Design (SID) is a computer-aided visual facility for hierarchical (or recursive) design of complex systems. SID is built as a provision to make the potential of our graph theoretical design tool RGF (the recursive graph formalism) actually available to system designers. RGF, as we initially proposed in 1978, aimed at providing a logical basis for interactive design evolution from global to detailed, and/or from simple to complex. RGF was actually applied to designs of hospital information systems and petrochemical plants, and was proven useful for logically detecting and preventing human design errors and for computer-aided design evolution. SID includes the capabilities of SARA of UCLA and SADT of SofTech which are known as the system specification methodologies based on hierarchically structured graphs. Related works using similar graphs in other areas are H-graphs in language and automata theory, and DRLH in artificial intelligence.

Basically, SID consists of an interactive computer graphics to display design specifications for designer’s visual inspection, a database system to verify, store, update, retrieve and control the design specifications in a shared file, and a design processor to execute design operations. In the databases, both design specifications and design operators are stored to allow sharing of frequently used designs, and design processes as well. Another major task of the current version of SID is to provide system developers with design evaluation facilities. The motivation for this is to cut down the system development cost by precluding the chances of implementing “poor” (i.e., would be marked “poor” if evaluated) designs.

DESIGN OF SYSTEM STRUCTURES AND PROPERTIES

Any system design usually consists of the specifications of the structures and properties of a system. The structure describes the system organization representing its subsystems (also called components, parts, or elements), environments, interfaces (also called ports, gates or terminals) and their relationships. The properties describe various information, associated with the system and its structure. A graphical representation of the design specification is called a design schema. The system components, environments and interfaces can have their own structures and properties, and hence, for the sake of generality, can be recursively viewed as systems again.

As a matter of fact, designer’s recursive view can go two ways—from general to special and from special to general. More precisely stated, when the structure of the system is designed, any other systems related with it are classified into three system categories depending on their relationship types: (1) system category “environment,” when the relationships are associations—that is, the systems are outside a given system; (2) system category “subsystem,” when the relationships are inclusions—that is, the systems are inside a given system; (3) system category “port,” when the relationships are interfaces—that is, the systems are on the boundary of a given system.

Once the designing of the structure of the system is done, the structure can be related with various records of information in a database as its properties. As usual in any database, each record consists of several fields. Depending on whether a record is related with a system or a relationship, it is called a system record or a relationship record. Therefore, the design of the system properties is a simple matter of relating the system structure with the database records.

RECURSIVE GRAPH FORMALISM (RGF)

The recursive graph formalism (RGF) is devised to combine the following two major potentials of widely used interactive design tools into one: (1) design visuality by displaying graphs for easy human understanding and inspection of hierarchical design evolution as typical in hierarchical structure diagrams and system charts; (2) design automation by machine processing, analysis and evolution of formally specified design as typical in various hardware design and program flow analysis methods.

RGF consists of recursive graphs (R-graphs) to represent design specifications, and recursive graph operators (R-operators) to manipulate them.

R-graphs

Traditionally, the most popular way of representing structures is by a graph, because it is both visual and formal. A
graph $G$ is simply a triple which consists of nodes $N$, arcs $A$, and an arc function $af$ specifying the ordered node pairs to which arcs are incident. Then,

$$G = (N, A, af)$$

where $af : A \rightarrow N \times N$.

Usually systems are represented by nodes, and their relationships by arcs. To utilize all the powers of mathematics, formalism and algorithms of graph theory, we extend graph theory to incorporate designer’s recursive view of the system structures and of the associated system records. First, an arc function $af$ is extended to map an arc to a pair of node subsets such that $af : A \rightarrow 2^N \times 2^N$. This extension is useful for a designer to relate groups of nodes by an arc. Next, a subnode function ($sn : N \rightarrow 2^N$), a port function ($pt : N \rightarrow 2^N$), and a subarc function ($sa : A \rightarrow 2^A$) are annexed to incorporate inclusion relationships among systems, interface relationships among systems, and inclusion relationships among associations, respectively. These extensions increase the structure representation capabilities of graphs. Further extension is done to give the property representation capabilities to graphs. That is, a node semantic function ($sn : N \rightarrow NR$) and an arc semantic function ($as : A \rightarrow AR$) are annexed to link nodes and arcs to node records $NR$ and arc records $AR$, respectively. Thus, an $R$-graph is given by:

$$R = (N, A, af, sn, pt, sa, ns, as).$$

The domains of the functions $af$, $sn$, $pt$, $sa$, $ns$ and $as$ are extended to incorporate the value “undefined (or under defined)” and the value “overdefined (or redundant).” The value “undefined” indicates that “the value is not available now but will come.” It is different from the “null” value which means “the fact that no value exists is made sure.”

Given a design schema, repeated applications of $sn$ and $pt$ to the nodes and of $sa$ to the arcs produce a hierarchy of the nodes and that of the arcs, respectively. We call them a node hierarchy and an arc hierarchy. Given a node or an arc of any hierarchy, the nodes or arcs produced earlier are called its ancestor nodes or arcs. It is reasonable to assume that in actual system design, any system cannot be subsystems (or ports) of two different other systems at the same time. Hence, in the rest of this paper, we only consider the case where the node hierarchies and the arc hierarchies form DAG (directed acyclic graphs). This actually increases the logical clarity of our formalism.

**Design schema base**

Two groups of records, one representing the structures and the other representing the properties of a system, are stored in a design schema base separately and as flat tables, so that both can be freely combined and utilized depending on application views. As a matter of fact, to substantiate this flexibility of table forms, we applied a pointer array technique to implement a database mapping. Figure 2 illustrates how a given R-graph is stored in a design schema base. In the structure base, node identifiers $NID$ and arc identifiers $AID$ are used as the primary keys of the tables which define functions $af$, $sn$, $pt$ and $sa$. In the semantic base, tuple identifiers $TID$ are the primary keys of node record $NR$ and arc record $AR$. We now can define the structure-semantic mappings, $ns$ and $sa$, as pointer arrays using only these identifiers. Hence the structure base and the semantic base can be updated as independently as possible. Please note that the meaning of symbols in Figure 2 are found in Figures 3 and 4.

**R-operators**

All the elementary operations to a node or an arc of a design schema are performed by recursive graph operators (R-operators). Such most primitive operations are collectively called an R-graph processor. Since design schemas are represented as flat tables, actually a table handler can be used as an R-graph processor. Throughout this paper, whenever necessary, R-operators are defined each time and used rather intuitively to increase the readability.

**DESIGN PROCESS FORMALIZATION FOR DESIGN PROCESS SHARING**

Design processes often contain similar and/or common basic processes, for example, to produce, integrate, reduce,
analyze and verify a design schema. In SID, a design course of a system interactively specifies a shareable design process as a sequence of R-operators. We call such a shareable design process also a design operator. Actually, these processes are stored in a design process base for a designer's later use and/or for sharing them among designers. They are driven through an interactive design command, issued by a designer. Combined with the design schema base, this design process base provides a flexible and powerful graphics-oriented tool for designers.

In the rest of this paper, we show examples of design processes to be shared. We also show that these design processes are actually representable in a common formalism of R-operators. For illustration, a well-known case of designing a pipeline system is examined. A pipeline system is a system which inputs a text from a card reader, formats it, and outputs it to a line printer. The text is formatted so that each text begins and ends with a blank page, each page begins and ends with a blank line, and each line is surrounded by blank margins. There, inputting, formatting and outputting a text are processed concurrently.

The global architectural design of the system is represented in Figure 5. The symbols used in this figure are explained in Figures 3 and 4. "Type Name" lists the reserved types of systems or of associations, "Graphic Symbol" is for display, and "Meaning" is for annotation.

### Figure 2—Recursive graph components.

### Figure 3—Reserved system types for concurrent system design.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Graphic Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-flow</td>
<td><img src="data_flow" alt="" /></td>
<td>Primitive active system P reads data D1 and writes data D2.</td>
</tr>
<tr>
<td>Control-flow</td>
<td><img src="control_flow" alt="" /></td>
<td>Activation of primitive active system P1 proceeds that of primitive active system P2.</td>
</tr>
<tr>
<td>Routine-flow</td>
<td><img src="routine_flow" alt="" /></td>
<td>Routine R defined as the destination of a routine-flow is invoked at the origin of the flow.</td>
</tr>
<tr>
<td>Control-link</td>
<td><img src="control_link" alt="" /></td>
<td>A control-link indicates existence of control-flows from ports of communication box C1 to those of communication box C2.</td>
</tr>
<tr>
<td>Use-link</td>
<td><img src="use_link" alt="" /></td>
<td>A use-link indicates existence of use-flows from the inside of routine R to the inside of machine M2, and represents that routine R uses routine R2.</td>
</tr>
<tr>
<td>Hierarchy-link</td>
<td><img src="hierarchy_link" alt="" /></td>
<td>A hierarchy-link indicates existence of use-links from routine R to the inside of machine M1 to the inside of machine M2, and represents that M1 and M2 are implemented by using routines of M2.</td>
</tr>
</tbody>
</table>

### Figure 4—Reserved association types for concurrent system design.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Graphic Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine</td>
<td><img src="machine" alt="" /></td>
<td>A machine represents a system which performs multiple functions and consists of routines performing those functions and data for communication among those routines.</td>
</tr>
<tr>
<td>Routine</td>
<td><img src="routine" alt="" /></td>
<td>A routine represents a system which performs a single function.</td>
</tr>
<tr>
<td>Communication box</td>
<td><img src="communication_box" alt="" /></td>
<td>A communication box represents a structured interface for communication among routines.</td>
</tr>
<tr>
<td>Routine-call</td>
<td><img src="routine_call" alt="" /></td>
<td>A routin-call represents an invoking of a routine.</td>
</tr>
<tr>
<td>Assignment</td>
<td><img src="assignment" alt="" /></td>
<td>An assignment represents a sequence of expression evaluations and data transfers.</td>
</tr>
<tr>
<td>Case</td>
<td><img src="case" alt="" /></td>
<td>A case represents a multiway control branch by a selector expression evaluation.</td>
</tr>
<tr>
<td>Token holder</td>
<td><img src="token_holder" alt="" /></td>
<td>A token holder represents a place in Petri net model.</td>
</tr>
<tr>
<td>Control port</td>
<td><img src="control_port" alt="" /></td>
<td>A control port is a port for control entry and exit of a macro active system.</td>
</tr>
<tr>
<td>Fork, Join</td>
<td><img src="fork_join" alt="" /></td>
<td>A fork and a join represent a transition having only one incoming and one outgoing s-flow, respectively, in Petri net model.</td>
</tr>
<tr>
<td>Data storage</td>
<td><img src="data_storage" alt="" /></td>
<td>A data storage represents a system holding some data.</td>
</tr>
<tr>
<td>Parameter</td>
<td><img src="parameter" alt="" /></td>
<td>A parameter represents a parameter of a machine or a routine.</td>
</tr>
</tbody>
</table>
Design integration/reduction operators

Here, very often used design processes for hierarchical design evolution, such as design integration, design abstraction and design reduction, are considered. We show that they actually can be defined in a form, applicable to general design schemas, and hence shareable among the designers.

Suppose a designer designs an input/output buffer "io-buffer." It is intended to be used as a common design schema of both the "inbuffer" and the "outbuffer" subsystems of a pipeline system which is shown in Figure 5. Figure 6 contains design schemas R2-R6 used in the following discussions. They are all specified in a shareable design form of R-graphs and stored in a design schema base. The designer starts to identify, as the subsystems of "io-buffer," the shared data storage "contents" and two routines "receive" and "send" which operate in parallel, communicating through the c-link "synchronize" as shown in design schema R2. Next, the designer refines "receive," by specifying its data flow as shown in R3 and also its control flow as shown in R4. Thus, the phase of designing the "io-buffer" subsystem is completed.

Now the design gets into the phase of subsystems integration. First, the designer produces the fully specified design schema of "receive" as R5. Actually, this design process is defined as an operation of integrating R3 and R4 by merging their common parts. In this case, the parts to be merged are two node pairs. One pair is ("receive" in R3, "receive" in R4). The other pair is ("text: = contents" in R3, "text: = contents" in R4). We specify the integrating process as a JOIN operation:

\[ R5 = \text{JOIN}(R3, R4, \{(\text{"receive" in R3, "receive" in R4}),\}, \phi) \]

where \( \phi \) denotes an empty set (in this case, of arc pairs). This is just an instance of a general form

\[ r_1 = \text{JOIN}(r_2, r_3, SN, SA) \]

where \( r_2 \) and \( r_3 \) are design schemas to be joined, \( r_1 \) is the resulting design schema, and \( SN \) and \( SA \) are lists of node pairs (including ports pairs) and arc pairs, respectively, to be merged. Since general forms are obvious from their instances, only instances are shown in the following discussions.

Next, the designer inserts schema R5 into the "receive" node of R2 by matching port pairs ("rsyn" in R2, "rsyn" in R5) and ("contents" in R2, "contents" in R5), and produces R6. This process is also defined as a ZIN (zoom-in
Figure 6—Design schemas appearing during the iobuffer design process.

node) operation as follows:

\[
R6 = \text{ZIN}(R2, R5, \{"rsyn" \text{ in } R2, "rsyn" \text{ in } R5, "contents" \text{ in } R2, "contents" \text{ in } R5\})
\]

This completes the subsystems integration phase. So far, we have identified two general integration operators, JOIN and ZIN.

We now consider a case where the designer likes to see an abstract (i.e., global) structure of a given schema. For example, in schema R6, the designer wishes to hide the uninterested details of "receive." This is done by applying ZON (zoom-out node) operation to R6 to obtain more global schema R2:

\[
R2 = \text{ZON}(R6, "receive")
\]

conversely, the designer can extract the interested inside detail of "receive" by EXN (extract node) operation:

\[
R5 = \text{EXN}(R6, "receive")
\]

Furthermore, if the designer wishes to analyze R5 from the view point of data flows, he or she uses SEL (select) operation for gathering the related nodes {"receive", "text", "text: = contents", "contents"} and arcs {"I", "m"}, and gets R3:

\[
R3 = \text{SEL}(R5, \{"receive", "text", "text: = contents", "contents"\}, \{"I", "m"\})
\]

or uses DEL (delete) operation for removing some nodes and arcs as the unnecessary details of R5:

\[
R3 = \text{DEL}(R5, \{"B", "E", "rsyn", "i", "o", "j", "F", "J", "F"\}, \{"a", "b", "c", "d", "e", "f", "g", "h"\})
\]

Design survey operators

A survey operator is a typical example to see how a design operator can be implemented by using R-operators and other design operators. For the detailed definition of the survey operator as a sequence of R-operators, please refer to Appendix 2.

A survey operator produces a design subschema by surveying a given design schema from a specific point of view. We now define a general survey operator \(\text{SURV}(R, NC, AC, SC)\) to search any given design schema \(R\) so that a designer can find such portions of \(R\) that satisfy his or her point of view. A general way of specifying a point of view is as a set of search conditions. In this case, they are \(NC, AC\) and \(SC\). We give here a little bit of terminological preparations to explain search conditions. A set variable is a variable which takes a subset of values. A variable to represent a field of a database record is an example of it. A field name variable is a variable which takes a field name as its value. We are now ready for explaining search conditions.

The first search condition \(NC\), called a node search condition, is in a form

\[
NC \land \ldots \land NC_m
\]

where \(NC_j = (NKEY_j of NFIELD_j)
\]

\(NKEY_j\) is a set variable, and \(NFIELD_j\) is a field name variable. \(NC\) indicates to select nodes if and only if their record contains one of the search key \(NKEY_j\) in the field \(NFIELD_j\). The second search condition \(AC\), called an arc search condition, is defined in the same way with \(NC\), by using field name variables \(\{AFIELD_1, \ldots, AFIELD_n\}\) and set variables \(\{AKEY_1, \ldots, AKEY_n\}\). The third search condition \(SC\), called a structure search condition, is in a form (struct by \(SK\)), such that a set variable \(SK\) \(\subseteq\) \{"NODE", "ARC"\}. "NODE" and "ARC" are literal constants, which, if appear in a given \(SK\), indicate to select all the ancestor nodes of nodes satisfying the search condition \(NC\) in the node hierarchy, and to select all the ancestor arcs in the same way for \(AC\), respectively.

Examples of quite commonly taken viewpoints are a "flow of control" and a "flow of data." As search conditions, we need to use only arc search conditions with "c-flow" for a control flow, and "d-flow" for a data flow. Thus, the data flow R3 and the control flow R4 of a given design schema R5 in Figure 6 are generated by simply applying SURV to R5 as follows:

\[
R3 = \text{SURV}(R5, \{"(d-flow)" of "(TYPE)"\}, \text{struct by \{"(NODE)"\}})
\]

\[
R4 = \text{SURV}(R5, \{"(c-flow)" of "(TYPE)"\}, \text{struct by \{"(NODE)"\}})
\]

where node conditions are null.
Design analysis and evaluation operators

Design analysis and evaluation operators are key tools which designers need to use in management of system development. In principle, application of them directly to a given design schema gives us the results of design analysis and evaluation. Experiences of systems development tell us that modularity is one of the most important common factors when evaluating various designs. The better the modularity of a design, the smaller the range of ripple effects of the modification causes. Therefore, when maintaining or adapting the design to application changes, good modularity certainly localizes the ripple effects of modification, thus decreasing the maintenance cost. Early analysis and evaluation of a design by modularity or any other measures have another significant managemental meaning. As mentioned before, it also certainly decreases the life cycle cost of the system being developed. This is because system developers can avoid implementing designs which are evaluated as "poor." As a case study, we illustrate in the following the use of SID, especially RGF, for defining a modularity analysis and evaluation operator.

Modularity

Modularity of a set of systems defined by Myers is known as an effective measure of ripple effects when one of the systems is modified. We show here that his definition can be formulated based on RGF and then can be evaluated automatically. As an example, we apply the results to evaluate the modularity of R7 in Figure 7. R7 is the design schema of the formatting subsystem of a pipeline system.

Suppose systems are connected only through shared data. Such data sharing is classified into three types specified by the following three predicates. Note that in the rest of this paper "the system represented by node x" is simply called "system x", and "A is defined by B" is denoted as "A = B"; (1) share(x, y) if system x and system y have access to common data d; (2) consist(x, y) if system y has access to data d which is a component of system x; (3) pass(x, y) if system x invokes system y passing at least one parameter. The above predicates are formulated in Appendix 1 based on RGF.

Modularity operator MDL(R) for design schema R first identifies all these relationships among systems \{x_i\} such that x_i \in N and \( \text{stype}(x_i) = \text{"machine"} \) or \text{"routine"}, and represents them in an \( n \times n \) square connectivity matrix \( C = (C_{ij}) \), where the i\text{th} column and j\text{th} row represent system \( x_i \) and \( x_j \) respectively, and \( n \) is the number of systems \( \{x_i\} \). Each matrix entry \( C_{ij} \) is defined such that

\[
C_{ij} = \begin{cases} 
  t_{ij}, & \text{if } i = j \text{ then 1 else 0;} \\
  s_{ij}, & \text{if } \text{share}(x_i, x_j) \text{ then 1 else 0;} \\
  c_{ij}, & \text{if } \text{consist}(x_i, x_j) \text{ or consist}(x_j, x_i) \text{ then 1 else 0;} \\
  p_{ij}, & \text{if } \text{pass}(x_i, x_j) \text{ or pass}(x_j, x_i) \text{ then 1 else 0.}
\end{cases}
\]

where

\[
t_{ij} = 1 + S + C + P
\]

\[
s_{ij} = 1 + S
\]

\[
c_{ij} = 1 + C
\]

\[
p_{ij} = 1 + P
\]

In the above, \( S, C, P, S, C, P \) are the probabilities that the modification of system \( x \) causes the modification of system \( y \) which is connected to system \( x \) through data sharing specified by share, consist and pass predicates, respectively. Myers' estimated values of these probabilities are as follows: \( S = 0.7, C = 0.6 \) and \( P = 0.2 \).

For example, the connectivity matrix of R7 is presented in Figure 8. Using such a connectivity matrix, modularity MDL(R) is defined such that

\[
\text{MDL}(R) = \frac{\sum_{i=1}^{n} \sum_{i=1}^{n} C_{ij}}{n}.
\]

MDL(R) means the expectation of the number of the systems which must be modified when one of the systems \( \{x_i\} \) is modified. Now, MDL(R7) is evaluated by using Myers' values as follows:

\[
\text{MDL}(R7) = \frac{17 + 0.59 S + 0.71 C + 0.82 P}{17} = 1 + 0.59 S + 0.71 C + 0.82 P = 2.0
\]

CONCLUSION

The capabilities of a System for Interactive Design, SID, currently tested at the University of Tokyo, were demonstrated. Their features are summarized in two points: "logical exactness"-"flexibility" combination, and "design"-

Figure 7-A design schema representing Hansen's design of the formatting subsystem of a pipeline system.
"design process" sharing. Shared design processes, called design operators, included design integration, abstraction, analysis and evaluation operators. It was a realization of our recursive design methodology RGF based on an extended graph theory. For illustration, some results of its applications to concurrent system design were also given.

ACKNOWLEDGMENT

The support of the JIPDEC, under Grant "5th Generation Computers," is gratefully acknowledged. Further support was provided by the MESC Software Engineering Project. Authors also wish to thank Miss M. Onoda, Dr. N. Ohbo, Messers. H. Kitagawa and A. Urabe for help in preparing this paper.

REFERENCES


APPENDIX I

The predicates share(x, y), consist(x, y) and pass(x, y) can be formulated based on RGF as follows:

\[
\text{share}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } x = y \land (\exists \text{deN}(\text{brother}(x, d) \land \text{channelaccess}(x, d) \lor \text{brother}(y, d) \land \text{channelaccess}(y, d)) \\
\text{FALSE} & \text{otherwise.}
\end{cases}
\]

\[
\text{consist}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } y \text{rs}(x) \land (\exists \text{deN}(\text{dech}(x) \land \text{channelaccess}(y, d)) \\
\text{FALSE} & \text{otherwise.}
\end{cases}
\]

\[
\text{pass}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } (\exists a \text{Ax}(a))(a) = ([x], [y]) \\
\text{NIL} & \text{otherwise (NIL stands for an undefined value)},
\end{cases}
\]

\[
\text{brother}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } \text{parn}(x) = \text{parn}(y) \\
\text{FALSE} & \text{otherwise,}
\end{cases}
\]

\[
\text{style}(x) = \text{a value of the type field of system record } ns(x),
\]

\[
\text{chn}(x) = \text{sn}(x) \text{Upt}(x),
\]

\[
\text{parn}(x) = \begin{cases} 
\text{NIL} & \text{otherwise (NIL stands for an undefined value)},
\end{cases}
\]

\[
\text{share}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } x = y \land (\exists \text{deN}(\text{brother}(x, d) \land \text{channelaccess}(x, d) \lor \text{brother}(y, d) \land \text{channelaccess}(y, d)) \\
\text{FALSE} & \text{otherwise.}
\end{cases}
\]

\[
\text{consist}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } y \text{rs}(x) \land (\exists \text{deN}(\text{dech}(x) \land \text{channelaccess}(y, d)) \\
\text{FALSE} & \text{otherwise.}
\end{cases}
\]

\[
\text{pass}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } (\exists a \text{Ax}(a))(a) = ([x], [y]) \\
\text{NIL} & \text{otherwise (NIL stands for an undefined value)},
\end{cases}
\]

\[
\text{brother}(x, y) = \begin{cases} 
\text{TRUE} & \text{if } \text{parn}(x) = \text{parn}(y) \\
\text{FALSE} & \text{otherwise,}
\end{cases}
\]

\[
\text{style}(x) = \text{a value of the type field of system record } ns(x),
\]
**APPENDIX 2**

Survey operator SURV(R,NC,AC,SC) is defined as a sequence of R-operators in the following.

procedure SURV(R,NC,AC,SC)
/*comment
NC:=(NKEY of NFIELD1)\ldots(NKEYm of NFIELDm),
AC:=(AKEY of AFIELD1)\ldots(AKEYn of AFIELDn),
SC:=(struct by SKEY),
X0,X1,\ldots,Xm,SX: a node set variable,
Ao,A1,\ldots,An,SA: an arc set variable,
begin SURV
STEP1: Find all arcs {a} of design schema R, and set Ao:=\{a\} and i:=0.
STEP2: Repeat [select from Ai all arcs {a} such that the arc record of arc aeA, has a value in a given search key AKEYi+1 at the field AFIELDi+1, set Ai+1:=\{a\} and increment i by 1] until i=n.
STEP3: If "ARC" e SKEY then find all ancestor arcs {a} of arcs An in the arc hierarchy and set SA:=\{a\} else SA:=\phi.
STEP4: Find all nodes {x} connected by arcs AnUSA, set X0:=\{x\} and i:=0.
STEP5: Repeat [select from Xi all nodes {x} such that the node record of node xeX, has a value in a given search key NKEYi+1 at field NFIELDi+1, set Xi+1:=\{x\} and increment i by 1] until i=m.
STEP6: If "NODE" e SKEY then find all ancestor nodes \{x\} of the nodes Xm in the node hierarchy and SX:=\{x\} else SX:=\phi.
STEP7: Execute SEL(R,XmUSA,XmUSA).
end SURV

Note that for a given set S, if variable v is set variable ve2^i, while veS if v is a (usual) variable.