TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

An implementation model for GOOD

Citation for published version (APA):
Reus, de, D. (1991). *An implementation model for GOOD*. (Computing science notes; Vol. 9105). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1991

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 02. Dec. 2021

An Implementation Model for GOOD

by

Dick de Reus

91/05

April, 1991

COMPUTING SCIENCE NOTES

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.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

An Implementation Model for GOOD

Dick de Reus
Department of Mathematics and Computing Science
Eindhoven University of Technology
Eindhoven, the Netherlands

Abstract

In this paper, algorithms are given to derive string representations for GOOD graphs. Instance graphs are represented by strictly formatted strings, and schema graph representations form grammars that generate those kind of strings. Finally, a meta grammar is given, generating these kind of grammars. Thus, the meta grammar forms a grammar implementation model for GOOD. This model may also function as a basis for the definition of constraints in GOOD, implemented as attributes added to the grammar.

1 Introduction

In recent years, data structuring is tending towards *object* or data modeling, rather than *relation* or structure modeling. Since long, graphs are used to represent and model data *structures*. The Graph-Oriented Object Data model GOOD [1, 2] deals with both data and structure modeling, providing facilities to query, browse, update, and restructure model graphs.
Seen as an object-oriented data model, GOOD supplies a graph transformation language that is rather powerful, and that provides enough operations to define querying, browsing, updating, and restructuring of data models in terms of graphs [2]. Because of its graph orientation, GOOD is suited for both study and development of graphic-oriented database end-user interfaces [1].
One topic of theoretical research is the addition of more semantics to data (structure) models, the first step of which will be done in this paper: we will develop an implementation model for GOOD in terms of grammars [3]. With this model, we can define attributes on grammars, representing constraints on data models.
The implementation model gives algorithms to represent two-dimensional data and structure graphs—two-dimensional because of the distinction between nodes (data) and links (structure)—in one-dimensional strings—one-dimensional because a string consists of words.

Figure 1 gives a schematic view of the topics discussed in this paper. The left hand side shows the relation between the GOOD model itself, GOOD schemas, and GOOD instances. The model defines an infinite set of schemas, each of which on its own defines a possibly infinite set of instances. We will give a graph-based definition of the GOOD model in section 2 by giving definitions for GOOD schemas and GOOD instances. These definitions are given in terms of graphs. In section 3, we will give a transformation $\mathcal{R}$, mapping instances I to

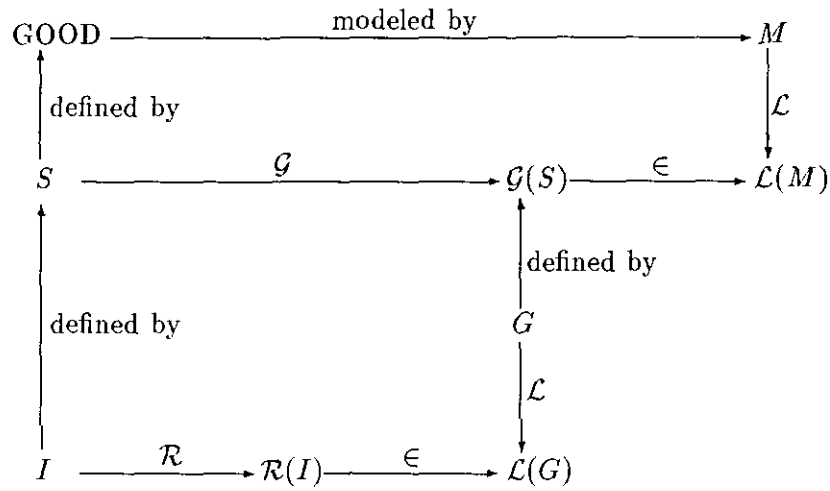

Figure 1: Schematical outline of this paper.

string representations $\mathcal{R}(I)$. Analogously, we will give a transformation $\mathcal{G}$ in section 4, mapping GOOD schemas S to grammars $\mathcal{G}(S)$. The language $\mathcal{L}(G)$ of a grammar G is defined as the set of all strings generated by that grammar. We will see that instance representations generated by $\mathcal{R}(I)$ are elements of the language of $G(S)$, for I's defined by S in the GOOD model. Furthermore, we will model the GOOD model by a meta grammar M, the language of which contains all schema grammars. This will be done in section 5. The final section gives some topics for further research.

2 The GOOD model

The object data model GOOD distinguishes between object base *schemas* and object base *instances*. An object base schema defines a set of object base instances.
The use of graphs in modeling data can be restricted at two levels: the model level and the schema level. The GOOD *model* restricts the use of graphs by demanding that a GOOD graph is directed; that there are two kinds of nodes, printable and non-printable; and that there are two kinds of edges, functional and non-functional. Furthermore, nodes and edges have labels, and printable nodes have an additional print label.

2.1 Object base schemas

The division of node labels and edge labels in the model is projected on GOOD *schemas*: each schema defines sets of node labels and edge labels. Thus, a schema further restricts the use of GOOD graphs by demanding that labels for nodes and edges should be picked from those sets, and by giving rules for combining edges and nodes.

Definition 2.1 (object base schema) *A GOOD* object base schema *or* structure graph *is a quintuple* S = (NPOL, POL, FEL, NFEL, $\mathcal{P}$), *with* NPOL *a finite set of non-printable*

object labels; POL *a finite set of printable object labels;* FEL *a finite set of functional edge labels; and* NFEL *a finite set of non-functional edge labels.* $\mathcal{P}$ *is a set of productions* (L, f) *with* $L \in$ NPOL *and* f : FEL $\cup$ NFEL $\rightarrow$ NPOL $\cup$ POL *a partial mapping from edge labels to object labels. Furthermore, there is a* print mapping π *defined on* POL, *such that for each printable node label* $n \in$ POL, $\pi(n)$ *gives a set of printable constants. A lexicographic ordering* $\preceq$ *is defined on edge labels* FEL $\cup$ NFEL.

We will represent object base *schemas* by graphs with two kinds of labeled edges. Figure 2 is an example schema graph. The set of all labeled boxes in a schema graph represent *NPOL* of a schema, the set of labeled circles represent *POL*, all labeled single-headed arrows represent *FEL*, and double-headed arrows represent *NFEL*. The production function $\mathcal{P}$ is represented by the placement of the arrows. In schema graphs, a node label will be used for one node only. The graph of figure 2 represents a schema $S = (NPOL, POL, FEL, NFEL, \mathcal{P})$ with:

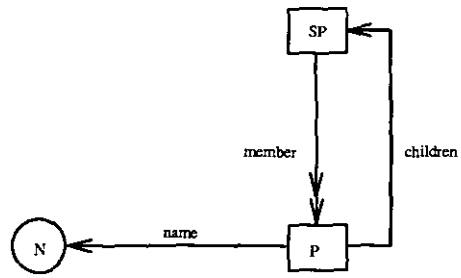

Figure 2: Family schema.

$NPOL = \{\texttt{P}, \texttt{SP}\}$;
$POL = \{\texttt{N}\}$;
$FEL = \{\texttt{children}, \texttt{name}\}$;
$NFEL = \{\texttt{member}\}$;
$\mathcal{P} = \{(\texttt{P}, \{(\texttt{children}, \texttt{SP}), (\texttt{name}, \texttt{N})\}), (\texttt{SP}, \{(\texttt{member}, \texttt{P})\})\}$.

Associated with this schema is the print mapping π with:

$\pi(\texttt{N}) =$ { "Brian", "Cindy", "Glenda", "Hester", "Jim", "Peter" }.

The interpretation of this schema is: A set of persons (SP) consists of several members of type person (P). A person has a name which is one of "Brian", "Cindy", etc., and a person has children forming a set of persons.
The schema tells us that printable nodes in an instance graph have label N, non-printable nodes have label P or SP, functional edges have label name or children, and non-functional edges have label member. It furthermore tells us that edges with label name only can start at nodes with label P and end at nodes with label N; edges with label children start at nodes with label P and end in nodes with label SP; and edges with label member start at nodes with label SP and end at nodes with label P. The print mapping π tells us that nodes with label N have an additional print label which is one of {"Brian", "Cindy", "Glenda", "Hester", "Jim", "Peter"}.

This schema specifies that there are three *types* of objects that can be represented in instances: objects of type SP, objects of type P, and objects of type N. Because N is a printable object label, objects of type N are *printable* objects, that is, they have a printable value. Furthermore, objects of type P can have two attributes, one called name, the other called children. These attributes are functional. Objects of type SP can have a (non-functional or set) attribute called member.

2.2 Object base instances

Associated with an object base schema S is a set of object base instances $\mathcal{D}(S)$. An *instance graph* I –also called *object base instance*– is a GOOD graph that complies to a GOOD schema:

Definition 2.2 (object base instance) *Let S = (*NPOL, POL, FEL, NFEL, $\mathcal{P}$*) be an object base schema with associated print mapping π. An* object base instance *or* instance graph *over S is a labeled directed graph $I = (N, E)$, with:*

- *N is a set of labeled nodes; for each $n \in N$, $\lambda(n)$ is its label which is in* NPOL $\cup$ POL. *n is called a* printable node *if $\lambda(n) \in$ POL; otherwise, n is called a* non-printable node. *($\lambda(n) \in$ NPOL).*
  Each printable node n of N is a triple $<$ id$(n), \lambda(n),$ print$(n) >$, with id(n) *a unique identifier, and* print$(n) \in \pi(\lambda(n))$. print(n) *is called the print label of n. Each non-printable node n is a pair $<$ id$(n), \lambda(n) >$, with* id*(n) a unique identifier.*
- *A total ordering $\leq$ is defined on nodes: for each pair of nodes $m, n \in N$, if $m \neq n$ then either $m \leq n$ or $n \leq m$.*
- *E is a set of labeled edges; the label $\lambda(e)$ of an edge $e \in E$ is an element of* FEL $\cup$ NFEL. *If $\lambda(e) \in$ FEL, e is called a* functional edge*; if $\lambda(e) \in$ NFEL, then e is called a* non-functional edge. *Each edge e is a triple $< m, \alpha, n >$, with m and n in N; m a non-printable node; and $\alpha = \lambda(e)$. e is called an* outgoing edge *for node m, and an* incoming edge *for node n. m is called the* start node *of e, n is called e's* end node.
- *For each edge $< m, \alpha, n >$ in E, there is a production $(\lambda(m), f)$ in $\mathcal{P}$ such that $f(\alpha) = \lambda(n)$.*
- *Functional edges are unique with respect to a node. That is, for each pair e_1, e_2 of functional edges with $e_1 =< m, \alpha_1, n_1 >$, $e_2 =< m, \alpha_2, n_2 >$; if $\alpha_1 = \alpha_2$, then $n_1 = n_2$.*

The set of instances $\mathcal{D}(S)$ belonging to an object base schema S is the set of all instances satisfying definition 2.2.
We will represent object base *instances* by graphs like we do for schemas. However, in an instance graph, labeled boxes represent non-printable objects, labeled circles denote printable objects, always having an additional print label (its value). Several nodes with the same label denote different objects, labeled arrows leaving a node represent the object's attributes.
In principle infinitely many instances are associated with the schema of figure 2. One example is the instance graph in figure 3. In this graph, there are five nodes with label P. Each one of

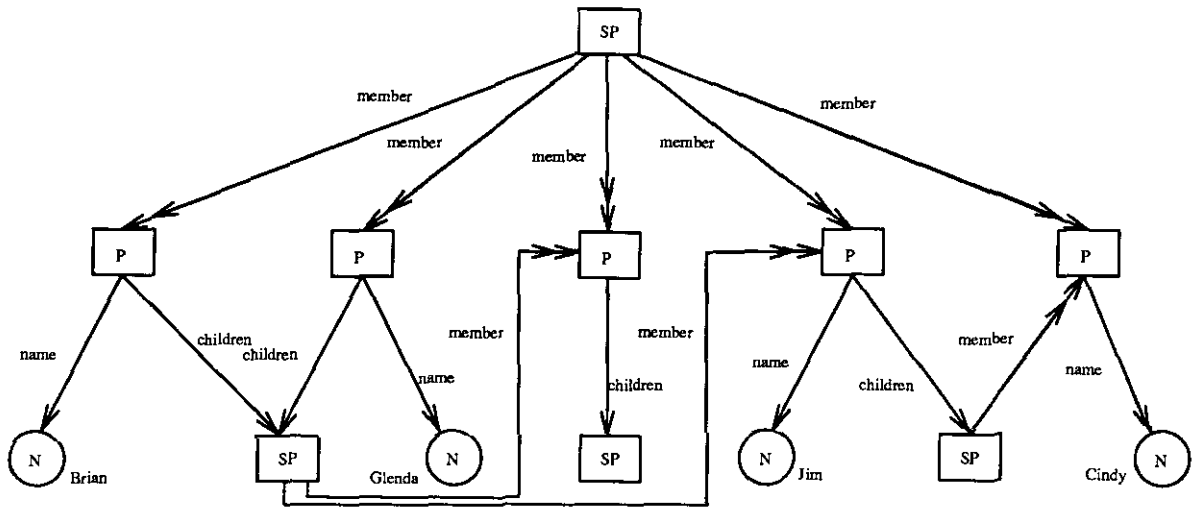

Figure 3: Family instance.

them represents a person. For some persons the name is represented, for some persons their children are represented as sets of persons. From this instance, we can deduce that "`Brian`"[1] and "`Glenda`" both have (the same set of) two children, one of them called "`Jim`". The other one's name is not represented in this model. "`Jim`" also has a child with name "`Cindy`".

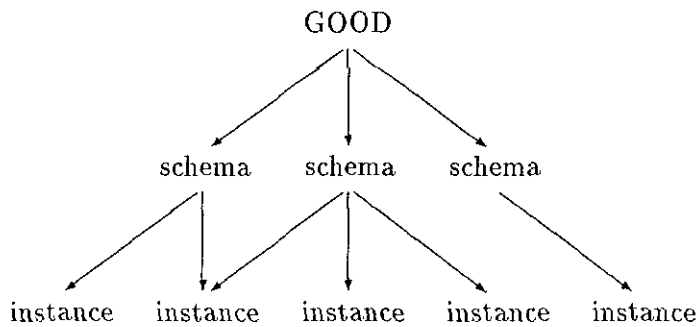

Figure 4: Relation between the GOOD model, schemas and instances.

In figure 4 we see an example of the relations between the GOOD model, GOOD schemas, and GOOD instances: the GOOD model defines how schemas look like, a GOOD schema defines a class of GOOD instances. Note that an instance graph can be an instance of several schemas.

[1] More formally spoken: an *object* with *object label* P, and with *functional attribute* labeled name with *print label* '`Brian`'.

3 An implementation model for GOOD instances

Apart from defining classes of schema graphs and instance graphs, GOOD supplies mechanisms to query, browse, restructure and update GOOD graphs. Each of these actions can be expressed in terms of graph actions defined in the GOOD *transformation language*. In the scope of this paper we will not go into details; more information can be found in [2] and [1].
One step in the process of interface development for GOOD is the development of a computer's internal representation of GOOD graphs in the form of strings. In this section we will define an implementation model for GOOD instances in terms of strings. We will first show how instance graphs can be represented in strings by giving a function which maps GOOD graphs to strings. Because in general one single graph can be represented by several strings, we will define equivalence classes of strings. By defining normal forms on strings, we can represent an equivalence class by one single (normal) string.
Next, we will define the class of GOOD instances belonging to a certain schema by deriving a grammar out of the schema which generates normal strings of all equivalence classes representing instances of the schema. This will be done in the next section.

3.1 Normal form

We will represent an instance graph by a list consisting of representations of the nodes of the graph. Each node is then represented by its identifier and a list of edges leaving the node. Each edge representation represents the edge's label and a reference to the edge's end node.[2] In this way, each item of the graph is represented exactly once.
In the sequel, we will use lists to represent information. If no confusion arises, we will denote a list $[e_1[e_2[\cdots[e_n[\ ]]\cdots]]]$ by $[e_1, e_2, \ldots, e_n]$. Note that, as a consequence of this, a list $[e[\ ]]$ with exactly one element is denoted by $[e]$.

We choose to represent each instance by a string in *normal form*, which we will define here. We will make use of an ordering on edges, defined in terms of lexicographic ordering of edge labels in the definition of a normal form for edge lists:

Definition 3.1 (edge ordering) *The relation $\leq$ between edges e and f with the same start node is defined by: $e \leq f \Leftrightarrow \lambda(e) \preceq \lambda(f)$, where $\preceq$ is lexicographic ordering.*

Note that, as a consequence of definition 2.2, $e \leq f \wedge f \leq e$ (that is, $\lambda(e) = \lambda(f)$) for the same start node implies that e and f are non-functional edges.
We will represent a list of edges leaving the same node in *normal form*:

Definition 3.2 (edge normal form) *An edge list $[e_0, \ldots, e_k]$ of edges with the same start node m is in* weak edge normal form *if and only if for each pair of edges e_i and e_j with $i \leq j$ it holds that $e_i \leq e_j$. It is in* strong edge normal form *if for each pair e_i, e_j with $i < j$ for which $\lambda(e_i) = \lambda(e_j)$ holds, the end nodes n_i of e_i and n_j of e_j are in relation $n_i \leq n_j$.*

[2] In fact, this is a reference to the end node's *representation*.

Thus, in an edge list in normal form, the edges are ordered according to their labels and their end nodes. According to definition 2.2, the edges leaving the same start node and having the same edge label, cannot have the same end node.
In the sequel, we speak of *edge normal form*, meaning strong edge normal form.
Nodes will also be represented in normal form, that is, as an ordered node list.

Definition 3.3 (node normal form) *A list of nodes* $[n_0, \ldots, n_k]$ *is in* node normal form *if and only if for each pair* n_i, n_j *with* $i \leq j$, *it holds that* $n_i \leq n_j$.

Thus, a node list in normal form is an ordered node list. If no confusion can arise, we will simply speak of *normal form* both for node lists and edge lists, meaning node normal form and (strong) edge normal form respectively.

3.2 Instance representation

According to definition 2.2, an instance *graph* consists of two sets: the set N of nodes, and the set E of edges. An instance *string* will consist of a *list* of node representations. In order to create a list representation of an instance, we have to *enlist* it. This is done by the mapping $\mathcal{E}$. In appendix A, the definition of $\mathcal{E}$ is given. It makes use of the edge enlist function $\mathcal{E}_e$ and the select function $\mathcal{S}$. The straightforward enlist function $\mathcal{E}_e{}'$ creates an edge list in normal form, in which each item has the form $< l, n >$, and in which several items may occur with the same label l. We will use a *contract* function $\mathcal{C}$ which shrinks an enlisted set of edges by representing all edges with the same label by one list item consisting of the label and a list in normal form of end nodes. In case there is only one item with a certain label, the only end node is given, that is: $\mathcal{C}$ has no effect. A list of edges `[<member, n0>, <member, n3>, <member, n4>, <name, n2>]` is contracted by $\mathcal{C}$ into the list `[<member, [n0, n3, n4]>, <name, n2>]`. The edge enlist function $\mathcal{E}_e$ now is composed of $\mathcal{C}$ and $\mathcal{E}_e{}'$.
The function $\mathcal{S}$ selects the edges with same start node n out of a set of edges. Selecting the edges leaving node `n1` from the set:

```
{<id(n0), name, n2>,
 <id(n1), member, n3>,
 <id(n1), member, n0>,
 <id(n4), name, n6>,
 <id(n1), name, n2>,
 <id(n1), member, n4>
}
```

results in the set:

```
{<id(n1), member, n3>,
 <id(n1), member, n0>,
 <id(n1), name, n2>,
 <id(n1), member, n0>
}
```

Thus, an instance $I = (N, E)$ is enlisted by $\mathcal{E}(I) = [m_0, \ldots, m_n]$ which is called a *node list.* Each item of that list is a *node description* consisting of a node identifier and a list of the node's edges.

Definition 3.4 (node identifier) *The* node identifier $i(n)$ *of a node* n *is its label* $\lambda(n)$, *associated with its* identifier name $\mathtt{id}(n)$. $\lambda(n)$ *is called the* identifier type.

We will associate an identifier type with an identifier name by connecting them by means of an underscore "_". Suppose we have a node n with identifier type $\lambda(n) = \mathtt{Empl}$, and identifier name n, then its node identifier $i(n)$ is $\mathtt{Empl\_}n$. The function $\mathcal{R}_n$ performs the representation of one node with its edges.
The function $\mathcal{R}_l$ transforms an edge enlisting into a list of edge representations.
An edge is represented by its label and the node identifier of its end node, or—in case it is a non-functional edge—a list of end node identifiers. $\mathcal{R}_e$ performs the transformation of an edge or group of edges with the same label into an edge representation.

Consider the instance graph in figure 3 again. The instance representation for this graph is:

```
[
<SP_a: [<member [P_b, P_c, P_d, P_e, P_f]>]>,
<P_b: [<children SP_h>, <name N_g>]>,
<P_c: [<children SP_h>, <name N_i>]>,
<P_d: [<children SP_j>]>,
<P_e: [<children SP_l>, <name N_k>]>,
<P_f: [<name N_m>]>,
<N_g: Brian>,
<SP_h: [<member [P_c, P_d]>]>,
<N_i: Glenda>,
<SP_j: []>,
<N_k: Jim>,
<SP_l: [<member [P_f]>]>,
<N_m: Cindy>
]
```

It is a list of node representations. Each node representation consists of a node identifier and an edge list. An edge list consists of edge representations. Non-functional edge representations contain node identifier lists. The node representation `<SP_h: [<member [P_c, P_d]>]>` consists of the node identifier `SP_h` and an edge list `[<member [P_c, P_d]>]`. This edge list has one element, `<member [P_c, P_d]>` which contains the node identifier list `[P_c, P_d]`.

We will denote the instance representation function $\mathcal{R}' \circ \mathcal{E}$ by $\mathcal{R}$. Note that we now have a method to define equivalence of two instance graphs I and I': they are equivalent if their representations $\mathcal{R}(I)$ and $\mathcal{R}(I')$ are.

4 An implementation model for GOOD schemas

Now that we have a uniform way to represent instance graphs, we would like to *construct* the class of instances belonging to a schema. We already have a definition of instances on a schema (definition 2.2), but it can be used only to verify whether a graph is an instance belonging to a certain schema. We do not have a *constructive* definition of instances. However, we can give such a definition for strings representing instance graphs, by defining the class of instance representations belonging to a certain schema by means of a *grammar*. In this section, we will give an algorithm to derive a grammar from a given schema. The grammar generates all strings that are representatives of instances of the schema.

4.1 Grammars

We will first introduce the concept of grammars.

Definition 4.1 (symbols, member, alternative)
A non-terminal symbol *is any sequence of characters, a* terminal symbol *is any sequence of characters between quotes.*
A member *is a terminal symbol or a non-terminal symbol.*
An alternative *is a list of members.*

The alternative `["BEGIN", body, "END"]` has three members, one of which is a non-terminal symbol, the others are terminal symbols.

Definition 4.2 (grammar rule) *A* grammar rule *consists of a left hand side (*lhs*) and a right hand side (*rhs*). The left hand side is a non-terminal symbol; the right hand side is a list of alternatives.*

We will denote a grammar rule by its left hand side followed by a colon and its right hand side; the alternatives of the right hand side are separated from each other by semicolons, and the right hand side is followed by a period. Thus,

```
greeting: "BEGIN", body, "END";
          single clause.
```

is another notation for the grammar rule
`<greetings, [["BEGIN", body, "END"], [single clause]]>`.

Definition 4.3 (grammar) *A* grammar *is a quadruple* $G = (N, T, \Sigma, R)$*, with* R *a list of grammar rules. For each non-terminal symbol* n *exactly one rule with left hand side* n *is in* R*. The first rule is called the* start rule*, its left hand side is the* start symbol Σ*.* N *is the set of non-terminal symbols;* T *the set of terminal symbols.*

N, T, and Σ can be derived from R: N is the set of left hand sides occurring in R, T is the set of non-terminal symbols occurring as members in the right hand sides of R, and Σ is the left hand side of the first rule of R. Therefore, we will represent a grammar by its rule list. Consider the following set R of grammar rules, defining the grammar $\mathcal{G}$:

```
greeting: "BEGIN", body, "END";
          single clause.

body: single clause, ";", single clause;
      single clause, ";", body.

single clause: "hello";
               "goodbye".
```

From this set, we can derive N by selecting all left hand sides, so N = {`greeting`, `body`, `single clause`}. The set of terminal symbols consists of all quoted strings, so T = {`"BEGIN"`, `"END"`, `";"`, `"hello"`, `"goodbye"`}. The start symbol is the left hand side of the first grammar rule: Σ = `greeting`.

The *language* defined by a grammar is the set of strings that are generated by that grammar. The set of strings defined by one grammar rule with left hand side l and right hand side r is the set defined by the alternatives of that rule. The language defined by the rule `single clause` of the grammar above, is {`"hello"`, `"goodbye"`}.
The set of strings defined by a list of alternatives is the set defined by the first alternative unified with the set defined by the other alternatives. The language defined by the right hand side of the rule for `greeting` is the union of the language defined by `"BEGIN", body, "END"`, and the one defined by `single clause`.
The set of strings defined by one alternative—a list of members—is the concatenation of the strings defined by each member.
The string defined by a terminal symbol is the symbol itself. The set of strings defined by a non-terminal symbol is the set defined by the rule with that symbol as right hand side.

Definition 4.4 (language) *The language $\mathcal{L}(G)$ of a grammar $G = (N, T, \Sigma, R)$ is the set of strings defined by the start rule.*

Note that the language of a grammar may be an infinite set. In fact, the language defined by the grammar $\mathcal{G}$ above is an infinite one:

```
L(G) = {"hello","goodbye",
        "BEGIN hello ; goodbye END",
        "BEGIN hello ; hello END",
        "BEGIN goodbye ; hello END",
        "BEGIN goodbye ; goodbye END",
        "BEGIN hello ; hello ; goodbye END",
        ⋮
       }
```

4.2 Schema representation with grammars

We will now generate a list of grammar rules from a GOOD schema. The generation algorithm can be found in appendix C. The list consists of several main parts. The *initial rules* represent the overall structure of an instance representation. The right hand side of the *node enumeration rule* enlists all alternative node descriptions. For each node label, a set of four *node specific rules* is generated. For printable node labels, the node specific rules contain a grammar rule representing π; for each non-printable node label, the production rules of $\mathcal{P}$ concerning that label are represented in a grammar rule. Finally, a set of *edge rules* is generated. Each edge rule represents an edge label and the end node's label.

initial rules The list starts with a rule that specifies that an instance is represented by a list of node descriptions:

```
instance: "[", node description list, "]".
```

The syntax of a node description list is described in the following rule:

```
node description list: "<", node description, ">";
                       "<", node description, ">", ",", node description list.
```

In appendix C, $\mathcal{I}$ represents the rule list containing these two initial rules.

node enumeration rule Furthermore, we need a grammar rule describing which kinds of node may appear. This rule enlists all possible node descriptions in an instance representation. In the family example of figure 2, this rule is:

```
node description: P description;
                  SP description;
                  N description.
```

The right hand side of the general form of this rule is given by $E_o(POL \cup NPOL)$, enumerating all object labels in non-terminal symbols.

node specific rules Grammar rules have to be generated for each alternative of `node description`. They are generated by $\mathcal{G}_{POL}$ for printable nodes, and $\mathcal{G}_{NPOL}$ for non-printable ones. In the family example, $\mathcal{G}_{POL}(\{\texttt{N}\})$ generates:

```
N description: N identifier, ":", "[", N, "]".
N: "Brian"; "Cindy"; "Glenda"; "Hester"; "Jim"; "Peter".
N identifier: "N", "_", identifier.
N identifier list: N identifier;
                   N identifier, ",", N identifier list.
```

Each group of node specific rules contains the rules for identifiers and identifier lists. They are given by the functions i_0 and i_1 in appendix C.
For printable node labels, the node specific rules contain a rule that enumerates the print labels for nodes with that label. The set of print labels for printable nodes with label l is given by the print mapping π. The function $\mathcal{E}_G$ enlists this set.
The node specific rules for non-printable node labels contain a rule that enumerates all possible edges for nodes with certain non-printable node label. In the example, these rules are:

```
P: P name N, P children SP.
SP: SP member P.
```

In general, these rules consist of a left hand side that specifies the object label of the start node, and a right hand side specifying all possible edges. The production function $\mathcal{P}$ gives for each non-printable object label l a function which gives the node label of end nodes of edges starting at nodes with label l. The function $\mathcal{E}_s$ transforms such a function into a list of alternatives.
The list of rules generated for non-printable nodes is given by $\mathcal{G}_{NPOL}$.

edge rules Finally, a list of rules has to be generated which represents all possible edges occurring in a GOOD instance. All possible node–edge combinations are encoded in the production function $\mathcal{P}$, which is a function mapping node labels to edge functions. Each edge function is a set of pairs (e, n), with e an edge label and n the label of end nodes for edges with label e. The function $\mathcal{G}_f$ transforms an edge function into a list of grammar rules belonging to the same node label. In the example, the edge function of node label `P` is {(`name`,`N`),(`children`,`SP`)}, for `SP` the function is {(`member`,`P`)}. These functions are transformed into:

```
SP member P: ;
             "<", "member", "[", P identifier list, "]", ">".
P name N: ;
          "<", "name", N identifier, ">".
P children SP: ;
               "<", "children", SP identifier, ">".
```

Note that each of these rules contains an empty alternative, specifying that an edge with the appropriate label is not necessarily present on each node with the correct label.
The list of edge rules belonging to one node label l is obtained by $\mathcal{G}_f$. The list of all grammar rules representing $\mathcal{P}$ can be obtained by $\mathcal{G}_{\mathcal{P}}$.

Finally, the grammarization of a schema $S = (NPOL, POL, FEL, NFEL, \mathcal{P})$ with associated print mapping π is defined by $\mathcal{G}(S)$.

5 An implementation model for GOOD

In the previous section we have seen that a certain schema $S = (NPOL, POL, FEL, NFEL, \mathcal{P})$ can be represented in a grammar by the function $\mathcal{G}$. This function gives a list of grammar rules for S. We have also seen that such a list could be represented in a string, and that it represents the grammar correctly. In this section we will go yet another step further: we will generate all grammars representing GOOD schemas by a meta grammar M. This grammar will function as an implementation model for GOOD, because it specifies the same as does GOOD: what are GOOD schemas, and which instance graphs belong to which schemas.[3] The definition of M can be found in appendix D.
The grammar generated by M consists of several groups of rules. The first group is the same for each grammar: the grammar part specifying the initial rules $\mathcal{I}$ already mentioned in the previous section. Next, a rule enumerating all possible node specifications is needed. After that, there are node specific rules and edge rules. They are generated by the following rules:

```
schema grammar: initial rules,
                node enumeration rule,
                node specific rules,
                edge rules.
```

The initial rules are the start rule and a rule specifying the form of a node description list. The grammar rules generated by the initial rules only is:

```
instance: "[", node description list, "]".
node description list: "<", node description, ">";
                       "<", node description, ">", ",", node description list.
```

The node enumeration rule is the first one that is depending on the schema. In the example schema of section 2, the meta grammar rules `node enumeration rule`, `description member list` and `description member` generate:

```
node description: P description;
                  SP description;
                  N description.
```

The node description syntax rule describes how a node description of a node with certain label looks like: it starts with an identifier, followed by a colon, followed by the node's semantics between square brackets. In the family example one such rule exists for nodes with label `N`:

```
N description: N identifier, ":", "[", N, "]".
```

Note that the item `node label` in the rule `node description syntax` occurs three times, and that it is important that each time it occurs in one and the same node description rule

[3] The GOOD transformation language operators—querying, browsing, restructuring and updating—are not considered here.

it should be the same one. Therefore, we have to add a context condition specifying that **node label** is one and the same.
The rules of the schema grammar that specify node semantics depend on whether the node label is printable or non-printable. In case it is a printable label, the rule specifies all possible print labels for nodes with that label. In the family example there is such a rule for nodes with label N:

```
N: "Brian"; "Cindy"; ...; "Peter".
```

In case the node label is non-printable, the rule has to specify which edges may be connected to nodes with that label. In the example we have such rules for labels P and SP:

```
P: P name N, P children SP.
SP: SP member P.
```

The identifier rules specify how identifiers and identifier lists of nodes with certain node label look like. In the family schema we have for instance:

```
N identifier: "N", "_", identifier.
N identifier list: N identifier;
                   N identifier, ",", N identifier list.
```

The first rule specifies that an identifier of a node with label N starts with the node label and an underscore. The second rule specifies how a list of node identifiers with the same node label looks like. For the meta grammar rules **identifier rule** and **identifier list rule**, we must again be aware of the fact that the node labels in the schema grammar rules generated by these meta grammar rules should be the same per rule. This gives another context condition.
The context condition for **edge list** is that the node labels of edge specifications occurring first are the same in one edge list. In one **node specific group** of schema grammar rules, all node labels occurring there should be the same.
Edge rules all look the same: they only differ in the left hand side and in the description of end nodes. For the meta grammar rule **end node description**, we also have to specify the context condition that the node label occurring first in the edge specification of an edge rule should be the same as the one occurring in the end node description of the same edge rule.

6 Concluding remarks

In this paper, we have described how to represent GOOD instances by strings, and how such instance representations can be generated by schema grammars. Furthermore, we have given a function that derives a schema grammar from a GOOD schema. Thus, we obtained an implementation model for GOOD instances and schemas. Next, we gave a meta grammar that generates schema grammars. Because the meta grammar is context-free, it does generate grammars that do not represent correct GOOD schemas. The next step we have to take

therefore, is the addition of context conditions to the meta grammar, as sketched in section 5. After this has been done, the meta grammar forms an implementation model for the *static* part of GOOD: we did not yet consider the part of the GOOD model concerning querying, browsing, restructuring and updating—the *dynamic* part.

The implementation model for GOOD schemas can be extended with constraints. This can be done by representing them as context conditions in the schema grammar. It will be the major topic of future research.

References

[1] M. Gyssens, J. Paredaens, and D. van Gucht. A Graph-Oriented Object Database Model. In *Proceedings of the 1990 ACM SIGMOD Conference*, pages 417–424, 1990.

[2] M. Gyssens, J. Paredaens, and D. van Gucht. A Graph-Oriented Object Model for Database End-User Interfaces. In *Proceedings PODS 1990 Conference*, pages 24–33, 1990.

[3] D.E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2), 1968.

A Instance representation

The representation of a GOOD instance $I = (N, E)$ is defined by:

$\mathcal{R}(I) = \mathcal{R}'(\mathcal{E}(I))$

$\mathcal{R}'([n_0, \ldots, n_m]) = [\mathcal{R}_n(n_0), \ldots, \mathcal{R}_n(n_m)]$

$\mathcal{R}_n(< n, [\,] >) =$ <$i(n)$: `print`(n) >, if n printable, else $\mathcal{R}_n(< n, [\,] >) =$ <$i(n)$: []>
$\mathcal{R}_n(< n, l >) =$ <$i(n)$: $\mathcal{R}_l(l)$ >

$\mathcal{R}_l([e_0, \ldots, e_n]) = [\mathcal{R}_e(e_0), \ldots, \mathcal{R}_e(e_n)]$

$\mathcal{R}_e(< l,$ `id`$(n) >) = < l\ i(n) >$, for n a node;
$\mathcal{R}_e(< l, [$`id`$(n_0), \ldots,$ `id`$(n_m)] >) =$ <l $[i(n_0), \ldots, i(n_m)]$>, for $n_0, \ldots, n_m$ nodes.

$\mathcal{E}((\{n_0, \ldots, n_m\}, E)) =$ [hd,tl], with $\{n_0, \ldots, n_m\}$ a set of nodes, and E a set of edges, and
hd $=< n_i, \mathcal{E}_e(\mathcal{S}(n_i, E)) >$, tl $= \mathcal{E}((\{n_0, \ldots, n_m\} - \{n_i\}, E))$, and $\forall_j : e_i \leq e_j$.

$\mathcal{E}_e = \mathcal{C} \circ \mathcal{E}_e{}'$, with $\circ$ function composition.

$\mathcal{C}([< l_0, n_0 >, \ldots, < l_q, n_q >]) = [< l'_0, m_0 >, \ldots, < l'_p, m_p >]$, with $p \leq q$, $m_i = n_k$ for some k, or $m_i = [n_{j_1}, \ldots, n_{j_x}]$ and $\forall_{a,b \leq p} : a \neq b \Rightarrow l'_a \neq l'_b$.

$\mathcal{E}_e{}'(\emptyset) = [\,]$.
$\mathcal{E}_e{}'(\{e_0, \ldots, e_n\}) = [< l_i, n_i > \mathcal{E}_e{}'(\{e_0, \ldots, e_n\} - \{e_i\})]$, with $\forall_j : e_i \leq e_j$, and $e_j =< n, l_j, n_j >$ edges with start node n.

$\mathcal{S}(n, \{e_0, \ldots, e_m\}) = \{e_i \mid e_i =<$ `id`$(n), l_i, n_i >\}$, with $\{e_0, \ldots, e_m\}$ a set of edges.

B Language of a grammar

The language $\mathcal{L}(G)$ of a grammar $G = (N, T, \Sigma, R)$ is the set of strings defined by:

$\mathcal{L}(G) = \mathcal{L}_r(< \Sigma, r >)$, for the start rule $< \Sigma, r >$.

$\mathcal{L}_r(< l, r >) = \mathcal{L}_A(r)$, with $< l, r >$ a rule of R.

$\mathcal{L}_A([hd\ tl]) = \{\mathcal{L}_a(hd)\} \cup \mathcal{L}_A(tl)$, hd a member list, tl and $[hd\ tl]$ alternative lists, $\mathcal{L}_A([\,]) = \emptyset$.

$\mathcal{L}_a([hd\ tl]) = \mathcal{L}_m(hd) \circ \mathcal{L}_a(tl)$;
$\mathcal{L}_a([x]) = \mathcal{L}_m(x)$;
$\mathcal{L}_a([\,]) =$"".

$\mathcal{L}_m(n) = \mathcal{L}_r(< n, r >)$, for some rule $< n, r >$, $n \in N$;
$\mathcal{L}_m(t) = t$, with $t \in T$.

$\sigma_1 \circ \sigma_2 = \sigma_1\sigma_2$, if σ_1 and σ_2 are strings;
$\sigma_0 \circ \{\sigma_1, \ldots, \sigma_n\} = \{\sigma_0\sigma_1, \ldots, \sigma_0\sigma_n\}$, σ_0 a string;
$\{\sigma_1, \ldots, \sigma_n\} \circ \sigma_0 = \{\sigma_1\sigma_0, \ldots, \sigma_n\sigma_0\}$, σ_0 a string;
and $\{\sigma_0, \ldots, \sigma_m\} \circ \{\rho_0, \ldots, \rho_n\} = \{\sigma_0\rho_0, \ldots, \sigma_0\rho_n, \ldots, \sigma_m\rho_n\}$.

C Schema representation with grammars

The grammarization of a GOOD schema $S = (NPOL, POL, FEL, NFEL, \mathcal{P})$ with associated print mapping π is defined by:

$\mathcal{G}(S) = \mathcal{I} \diamond [< \texttt{node␣description}, E_l(POL \cup NPOL) >] \diamond \mathcal{G}_{POL}(POL) \diamond \mathcal{G}_{NPOL}(NPOL) \diamond \mathcal{G}_{\mathcal{P}}(\mathcal{P})$,
with $\diamond$ list concatenation;

$\mathcal{G}_{\mathcal{P}}(\{(l_0, f_{l_0}), \ldots, (l_n, f_{l_n})\}) = \mathcal{G}_f(l_0, f_{l_0}) \diamond \mathcal{G}_{\mathcal{P}}(\{(l_1, f_{l_1}), \ldots, (l_n, f_{l_n})\})$

$\mathcal{G}_f(l, \{(e_0, l_0), \ldots, (e_n, l_n)\}) = [e] \diamond \mathcal{G}_f(l, \{(e_1, l_1), \ldots, (e_n, l_n)\})$, with
$e =< l\texttt{␣}e_0\texttt{␣}l_0, [[\,], [\texttt{"<"}, \texttt{"}e_0\texttt{"}, l_0\texttt{␣identifier}, \texttt{">"}]] >$, if $e_0 \in FEL$;
$e =< l\texttt{␣}e_0\texttt{␣}l_0, [[\,], [\texttt{"<"}, \texttt{"}e_0\texttt{"}, \texttt{"[}", l_0\texttt{␣identifier␣list}, \texttt{"]"}, \texttt{">"}]] >$, if $e_0 \in NFEL$;

$\mathcal{G}_{NPOL}(\{l_0, \ldots, l_n\}) = [p(l_0), e(l_0), i_0(l_0), i_1(l_0)] \diamond \mathcal{G}_{NPOL}(\{l_1, \ldots, l_n\})$;
$e(l) =< l, \mathcal{E}_s(l, \mathcal{P}(l)) >$.

$\mathcal{E}_s(l, \{(e_0, r_0), \ldots, (e_n, r_n)\}) = [l\texttt{␣}e_0\texttt{␣}r_0, \ldots, l\texttt{␣}e_n\texttt{␣}r_n]$

$\mathcal{G}_{POL}(\{l_0, \ldots, l_n\}) = [p(l_0), v(l_0), i_0(l_0), i_1(l_0)] \diamond \mathcal{G}_{POL}(\{l_1, \ldots, l_n\})$;
$p(l) =< l\texttt{␣description}, [[l\texttt{␣identifier}, \texttt{":"}, \texttt{"[}", l, \texttt{"]"}]] >$;
$v(l) =< l, \mathcal{E}_G(\pi(l)) >$

$\mathcal{E}_G(\{p_0, \ldots, p_n\}) = [p_0, \ldots, p_n]$

$i_0(l) =< l\texttt{␣identifier}, [[\texttt{"}l\texttt{"}, \texttt{"\_"}, \texttt{identifier}]] >$

$i_1(l) =< l_{\sqcup}\texttt{identifier}_{\sqcup}\texttt{list}, [[l_{\sqcup}\texttt{identifier}], [l_{\sqcup}\texttt{identifier}, ",", l_{\sqcup}\texttt{identifier}_{\sqcup}\texttt{list}]] >$

$E_l(\{l_0, \ldots, l_n\}) = [["<", l_{0\sqcup}\texttt{description}, ">"]] \diamond E_l(\{l_1, \ldots, l_n\})$

$\mathcal{I} = [<\texttt{instance}, [["[", \texttt{node}_{\sqcup}\texttt{description}_{\sqcup}\texttt{list}, "]"]] >,$
$< \texttt{node}_{\sqcup}\texttt{description}_{\sqcup}\texttt{list}, [["<" , \texttt{node}_{\sqcup}\texttt{description}, ">"],$
$["<", \texttt{node}_{\sqcup}\texttt{description}, ">", "," , \texttt{node}_{\sqcup}\texttt{description}_{\sqcup}\texttt{list}]] >]$

D Meta grammar

```
schema grammar: initial rules,
                node enumeration rule,
                node specific rules,
                edge rules.

initial rules: start rule, node description list rule.
start rule: "instance: ""["", node description list, ""]"".".
node description list rule:
           "node description list: ""<"", node description, "">"";", newline,
           """<"", node description, "">"", "","", node description list.".

node enumeration rule: "node description:", description member list, ".".
description member list: description member;
                         description member, ";", description member list.
description member: node label, " description".
node label: identifier.

node specific rules: node specific group;
                     node specific group, node specific rules.
node specific group: node description syntax,
                     node description semantics,
                     identifier rules.

node description syntax: node label, " description: ",
                         node label, " identifier, "":"", ""["", ",
                         node label, ", ""]"".".

node description semantics: printable node description rule;
                            non-printable node description rule.
printable node description rule: node label, ":", print label list, ".".
print label list: print label;
                  print label, ";", print label list.
print label: string.

non-printable node description rule: node label, ":", edge list, ".".
```

```
identifier rules: identifier rule, identifier list rule.
identifier rule: node label, " identifier: """",
                 node label, """, """_""", identifier.".
identifier list rule: node label, " identifier list: ",
                      node label, " identifier;",
                      node label, " identifier, """,""", ",
                      node label, " identifier list.".

edge list: edge specification;
           edge specification, ";", edge list.
edge specification: node label, edge label, node label.
edge label: identifier.
edge rules: edge rule;
            edge rule, edge rules.
edge rule: edge specification, ": ""<"", """, edge label, """, ""[", ",
           end node description, ", ""]"", "">"".".
end node description: node label, " identifier";
                      node label, " identifier list".
```

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits.
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes.
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films.
85/04	T. Verhoeff H.M.L.J.Schols	Delay insensitive directed trace structures satisfy the foam the foam rubber wrapper postulate.
86/01	R. Koymans	Specifying message passing and real-time systems.
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specification of information systems.
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures.
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several systems.
86/05	J.L.G. Dietz K.M. van Hee	A framework for the conceptual modeling of discrete dynamic systems.
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP.
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers.
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987).
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language.
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing.
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86).
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes.
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4).

86/14	R. Koymans	Specifying passing systems requires extending temporal logic.
87/01	R. Gerth	On the existence of sound and complete axiomati zations of the monitor concept.
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases.
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems.
87/04	T.Verhoeff	Delay-insensitive codes - An overview.
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns - searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language.
87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts.
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas.
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples.

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces.
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen.
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations.
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG.
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence.
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems.
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples.
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to Handle Structured Information.
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems.
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication.
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way.
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification.
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling.
88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits.
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases.
88/12	A.E. Eiben	Abstract theory of planning.
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees.
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types.

88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations.
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors.
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems.
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a paralell program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.

89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.
90/1	W.P.de Roever-H.Barringer C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, pp. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.

90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.
90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses of "if..., then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.