The GOOD based hypertext reference model

Citation for published version (APA):

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.
Eindhoven University of Technology
Department of Mathematics and Computing Science

The GOOD Based Hypertext Reference Model

by

P. de Bra  G.J. Houben  J. Paredaens

Computing Science Note 91/26
Eindhoven, September 1991
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.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.
The GOOD Based Hypertext Reference Model

P. De Bra
G.J. Houben
Eindhoven Univ. of Technology
PO Box 513, NL-5600 MB
Eindhoven, The Netherlands
(+31 40 47.27.33)
(debra@win.tue.nl, houben@win.tue.nl)

J. Paredaens
University of Antwerp
Universiteitsplein 1
B-2610 Wilrijk, Belgium
(+32 3 820.24.01)
(pareda@cc.uia.ac.be)

Abstract

Most hypertext systems do not have a well-defined underlying data model. Some systems have a model, based directly on mathematics or on a specification language like Z or VDM. We propose a formal model for hypertext, based on the GOOD database model. Using this simple graphical database model we can cover all aspects of a hypertext, from the highest level (viewing the system as a graph of nodes and links) down to the lowest level (looking at individual bytes).

Operations that are performed on a hypertext, such as selecting an item on the screen, following a link, performing a keyword search, adding and removing links, defining and viewing contexts, etc., can all be described as GOOD queries and updates. Typical hypertext constraints, e.g. "links in a context can only connect nodes from the same context", can also be described in the same way. An additional advantage of the use of GOOD is that queries, updates and constraints are described in the same formalism as the overall structure of the hypertext: with directed labeled graphs and a query mechanism based on pattern matching.

Using the GOOD database model we describe all aspects of a hypertext. Doing so, not only provides a more thorough insight into the meaning of key concepts in hypertext, but also shows the purpose of using a graph-oriented database model for defining inherently object oriented database applications such as hypertext.
1. Introduction

Ben Shneiderman starts his (hyper)book [10] with the following definition of hypertext: “The most common meaning of "hypertext" is a database that has active cross-references and allows the reader to "jump" to other parts of the database as desired.” Despite this definition no well-known hypertext data model exists that is actually defined using a database model. From a database point of view a hypertext is very different from a traditional database. There is no well-defined structure consisting of a static set of relations, attributes and constraints. Furthermore, the data do not consist of small elements like numbers, dates, money figures and short strings. Text fragments, bitmaps or other graphical data, and other kinds of information one may find in a hypertext are difficult to store in relational databases. To model the constantly changing structure of a hypertext a different database model is needed, with explicit support for complex objects and a frequently changing database scheme. Hence our decision to use an object oriented database model.

A hypertext is typically represented in some graphical way. It consists of nodes and links which together form a graph. Because the relational model is ill-suited for representing this kind of data some authors defined new formal models, targeted explicitly towards hypertext or hypertext systems([2], [12], [7], [8], [3], [11]). The obvious disadvantage of inventing a special purpose (mathematically defined) model is that one cannot reuse the years of research that have been invested in database theory. To the best of our knowledge, no hypertext model exists which is defined using a general-purpose database model. The common graphical representation of hypertext led us to define a reference model for hypertext using the Graph-Oriented Object Database Model (GOOD)([4], [5]), which uses a similar graphical representation for every database. Beeri and Kornatzky [1] developed a logical query language for hypertext which can be easily mapped into a graph-based representation (which is what they do in their paper, in order to describe how query answering works). In GOOD such a graph-oriented representation is part of the data model. Furthermore, by considering both nodes and links to be objects our object oriented database model can describe more details about the links than their model.

All existing reference models for hypertext assume that the basic building blocks, the nodes and links, are black boxes, of which no other information is known than what is described by some attributes. They have no single formalism covering the structure of the graph, the contents of the nodes, attributes and access and browsing methods, etc.

Using the GOOD database model we are able to describe all implementation-independent details of a hypertext, including the internal structure of nodes. This serves two purposes:
— it shows that hypertexts can be modeled completely as special purpose databases;
— it shows that the GOOD database model is highly suitable as the vehicle for describing detailed hypertext data models, because of its graph-based nature and its query system.

In Section 2 we briefly recall the basic concepts of the GOOD Database Model. Section 3 describes the global structure of the GOOD-based Hypertext Reference Model. In Section 4 we describe the elements of our reference model in detail. In Section 5 we give an example of how specific types of nodes can be described by extending the definition of nodes. In Section 6 we show how the same graphical formalism that was used to describe the basic elements and their constraints can also be used for browsing and querying a hypertext.
2. The GOOD Database Model

The GOOD database model was first introduced in [4], [5]. It describes database systems by means of directed graphs. All operations on the database can be expressed in a graphical way as operations on the graphs, by means of the GOOD data transformation language. Queries are performed by means of (a graphical kind of) pattern matching. Methods allow the creation of abbreviations for complicated queries and updates. (In GOOD these are usually called "macros"). In this section we recall the graphical representation of the most important elements of GOOD. Space limitations force us to be very brief. To fully understand this paper one should be familiar with [4] and/or [5] first.

At the instance level, a database is represented by a directed labeled graph. The nodes of this graph represent the objects of the database. We will call them objects to avoid confusion. We distinguish between printable and non-printable objects, which are depicted as circular and square boxes. Between the objects we have functional edges, depicted by a single headed arrow (→) and non-functional (or set-valued) edges, depicted by a double-headed arrow (↔). The edges represent properties of the objects. A non-printable object has one label, indicating the type of the object. A printable object has a second label, indicating the (printable) value of the object. The label of an edge indicates the type of the edge. An example of a GOOD instance is given below:

```
          S
         /  \
        /    \
   name    address
     |      |     S
     |      |     Mountain
     |      | Ave
     |      \n     |        S
       |      number
      |        N
      |      25
      |      \n      |     street
      |          S
      |       Woodland
      |    Road
      |      
      |        S
      |    number
      |          N
      |        street
      |          S
      |    name
      |          S
      |    person
      |          N
      |    address
      |        S
```

The database scheme underlying the instance is represented by a set of graph productions. They consist of nodes representing object-types and edges representing property-types. Instance graphs are built from building blocks (the graph productions), by pasting together a number of copies of (parts of) the productions, and adding values to the printable objects. (This is not the way it is defined in the original papers on GOOD, but proving that our definition is equivalent is beyond the scope of this paper.) A possible database scheme for the example instance is:

```
          S
         /  \
        /    \
   name    address
     |      |     S
     |      |     Mountain
     |      | Ave
     |      \n     |        S
       |      number
      |        N
      |      25
      |      \n      |     street
      |          S
      |       Woodland
      |    Road
      |      
      |        S
      |    number
      |          N
      |        street
      |          S
      |    name
      |          S
      |    person
      |          N
      |    address
      |        S
```

One can always take productions apart or paste them together, because in the instances one may use part of a production (as long as there is an object at both ends of every edge). This implies that one can add object-types and edges to the database scheme without affecting the current database instance. The process of pasting building blocks together (and also cutting them) is described by 5 operations: object addition and -deletion, edge addition and -deletion and abstraction. We shall describe these operations when we use them for describing hypertext.
3. Overview of the GOOD Reference Model

At the 1990 NIST workshop on Hypertext Standardization [3], [7], [8] tried to propose a way to partition a hypertext (system) into conceptually different classes or layers of elements of which they describe a few in some mathematical language (like Z and VDM). Most layers are not included in the formal part of these models, e.g. contents and/or structure of an information node, browsing and querying facilities and user-interface aspects. They focus only on the structure of the hypertext as a network of nodes and links and possibly some higher level structures, a view which, in our opinion, is too narrow to be useful. Furthermore, they use specification languages which are not targeted towards database applications.

Somewhat like [7] we propose a set of layers describing different aspects of a hypertext system. The GOOD (-based hypertext) reference model includes more layers in the formal definition, hereby moving the formal description closer to the borderline between what can or cannot be described in an implementation-independent way. In Figure 1 we show the overall structure of a hypertext system. The dashed line separates the hypertext aspects (to the right of the line) from the system aspects. As a hypertext system is a special object oriented database system we have used the GOOD database model for this figure.

As can be seen from Figure 1 the structure of a hypertext database is described by a set of hypertexts.

Figure 1. Layers of a hypertext system.
Each hypertext consists of a set of contexts (not necessarily disjoint) containing nodes and links. This 3-level hierarchy of (hypertext) graph/context/nodes-and-links is borrowed from the HAM model [2].

Contexts are basically views of subgraphs of the hypertext. They can be used to identify private workspaces, sets of related nodes (e.g., the set of all examples, exercises, or all nodes belonging to a chapter of a book, etc.).

Links may exist between contexts, nodes and anchors or a combination of nodes and anchors as we will see later.

Nodes contain a set of bytes, in a way we will describe later, and also a set of anchors.

Anchors (often called buttons and fields in literature) are some sort of items which can be identified within a node. Usually an anchor is described by a small set of bytes, called the boundary of the anchor. In a text-node this could be the first and last byte of the anchor, in a bitmap it could be an outline, e.g., a rectangle or circle.

The elements of a hypertext system that are not formalized in our GOOD reference model, but that are assumed to be part of every implementation, are the database server, called "GOOD database system" in Figure 1, the output filters, used for displaying hypertext graphs, contexts, anchors, nodes, links and bytes, and a single input method for selecting a byte (i.e., a method which identifies the byte which is under the mouse cursor when the "select" button is pressed).

We use the GOOD query language to formulate constraints and queries. However, we do not describe the GOOD user-interface and database server. The implementation of GOOD will provide a graphical user-interface for drawing queries (as described in Section 6). Answers to queries will again be represented by graphs.

4. Basic Building Blocks

In this section we focus on the basic elements of a hypertext: the information nodes and the links connecting related nodes. We also describe contexts, which group nodes and links into higher-level entities.

- In most hypertext models (including HAM) a node is a black box containing arbitrary data. In the GOOD reference model we can open up this black box and describe its contents down to the level of individual bytes.

- A link defines a relationship between a source and a destination. In HAM [2] both the source and destination must be nodes. In [8] both must be a node or an anchor (or a set of nodes or anchors), while in [7] arbitrary complex nested structures are possible. In the GOOD reference model we have 2 kinds of links: links between contexts and links between nodes and/or anchors. The destination of a link may be a single context, node or anchor or a set of contexts, nodes or anchors.

- A context is a either a set of (sub)contexts and links or a set of nodes and links. Because contexts may contain other contexts hierarchical structures are possible.

- Attributes can be attached to contexts, nodes and links. They are used to describe additional properties of nodes and links.

In GOOD all objects have an object-identity (oid). However, this oid is "internal" and cannot be referred to in queries. Because we want to anticipate the integration of GOOD with external software we provide explicit external object identifiers for all our object types. This technique is often used when integrating object-oriented software with external applications.

In the sequel we shall use "typewriter" font when referring to words from GOOD figures, to make
sure that the distinction between the abstract concepts and the actual formal elements of the GOOD reference model is clear.

First we define the structure of a node. In general a node is an object with a contents which is a set of bytes, and an anchoring contents which is a set of anchors. A node also has an id, a set of attributes, and a set of display methods. In GOOD the structure of a node is described by the following production:

```
I/O display
node attributes
   contents
      id
   anchoring
      content
      byte N anchor
```

The information contained in a node is represented by a set of bytes. Depending on the kind (or type) of node additional information may be given. In the next section we describe the structure of a text-node. The reader may construct other types of nodes. (The GOOD reference model is open-ended.) The GOOD database model does not require completeness of the information. A node need not have attributes, contents or anchoring contents. A node must have an id however. In GOOD a constraint is described by means of a small GOOD program which identifies violations of the constraint. (The exact reaction of the system upon detection of a violation is not described here because that depends on the implementation of GOOD.) The GOOD program which identifies nodes that lack an id is given below:

```
NO

NO no-id node

NO no-id node id N
```

The first object with a dark border is an object addition. It creates a NO object. The second step of the program contains an edge addition. From every NO object (there is only one such object) to every node object an edge is created. The third step is an edge deletion. Every edge between a NO object and a node object that has an id is deleted. This leaves no-id edges between the NO object and all nodes which do not have an id.

Because the graphical representation of constraints is space-consuming we omit it for most constraints whenever the exact description in GOOD is fairly straightforward.

A link has a source and a destination. It also has attributes, display methods and an id. In HAM the only type of link is a link between nodes. Often however one needs to link a single word or phrase to a node or to part of a node, like for references, explanations, descriptions and examples. We prefer not to simulate these links by means of attributes but rather
by allowing the source and destination of a link to be an anchor as well as a node. One may also wish to define a link with multiple sources and/or destinations, but such a generalized link can easily be simulated by a set of "normal" links. The structure of a link is as follows:

A link must satisfy the following constraints:
- A link must have exactly one source and has a set of destinations.
- The source and destinations of a link must all be contexts or all not be contexts.
- The source and destinations of a link must belong to every context to which that link belongs (i.e. they must be a node or context belonging to the same contexts as the link or be an anchor of a node belonging to the same context).

The latter constraint is difficult to write down in the procedural GOOD query language. However, in [9] a rule-based graphical query language is given, called G-Log, in which this constraint can be described by a set of rules. A rule describing part of the constraint would look like the following:

The rule means that if the non-bold pattern is found (i.e. a link with a source context and a set of destination contexts), then the bold pattern must also exist (i.e. the source and the destinations must belong to the context contents of the context containing that link).

The higher level object type context and the lower level object types anchor and byte are described below.
Contexts must satisfy the following constraints:

- The context contents of a given context must not include that given context. Hence the context contents edges define a hierarchy on contexts.
- Either the context contents or the data contents of a context must be empty.

In addition to the constraints mentioned above a hypertext must satisfy the following constraints:
- Every context "belongs" to one or more hypertexts.
- Every node and link "belongs" to one or more contexts.
- Every anchor "belongs" to exactly one node.
- Every anchor is source or destination of at least one link.
- Every byte belongs to exactly one node.
- Every byte of the boundary of an anchor belongs to the node containing that anchor.

The basic building blocks and constraints defined above are sufficient to model hypertext. By explicitly including anchors and bytes the use of attributes is no longer necessary for describing the internal structure of a node. Hence attributes are only used to describe additional properties of a node. (The lack of anchors was a major drawback in HAM.) In the literature one often uses the word button to denote an an anchor which is the source of a link, and the word field to denote an anchor which is the destination of a link. Although a button is usually much smaller than a field there is no need for a conceptual difference; hence our common denominator anchor.

5. Types of nodes

The description of hypertext given in the previous section uses one and the same structure for all types of nodes. By adding additional information and/or constraints we can model different types of nodes. We shall only describe one type of node in this paper, namely text-nodes, but the technique used to describe text-nodes can be used to describe all types of nodes. The GOOD reference model is open-ended in the sense that one can add definitions of new types of nodes at one’s leisure.

The additional information of a text-node is described as a linked list of bytes. The start of the
list is connected to the node.

A text-node must satisfy the following constraints:
- Every byte of the node's contents must appear in the list.
- If the node's contents is not empty then the node must have a first byte.
- The list must be acyclic.

On a text-node we can define a "precedes" method describing the order of the bytes. This is similar to the implementation of the natural numbers in [5]. The method is a sequence of GOOD programming steps (written from left to right instead of top to bottom, to save space).

Because the precedes edge occurs in both the query part (non-bold) and the insertion part (bold) of the edge addition (in the third step of the program) the edge addition is recursive.

We shall use the precedes method in the queries in the next section, but we also need it to describe the additional structure of anchors:

An anchor in a text-node is determined by its begin and end byte.

An anchor in a text-node must satisfy the following constraints:
- The begin and end byte of an anchor must belong to the boundary of that anchor and no other bytes belong to that boundary.
- The begin byte precedes the end byte (in the contents of the node, which is represented as a linked list using the first and next edges).

In the next section we show that typical operations on text-nodes can indeed be performed with our definition of the internal structure of a text-node.

6. Queries on hypertext

In this section we illustrate that the GOOD query language is not only useful (and convenient) for describing hypertext but that it can also be used for typical browsing and querying operations.

A common action when using a hypertext system is clicking on a button (an anchor in our model). Selecting an anchor (which is the source of a link) tells the computer to follow the corresponding link, and to display the destination of that link. In our model there may be many
links originating from the same anchor, not just one. The only assumption we make about the user interface is that it is able to provide us with the id of the character (or byte) at which the mouse was pointing when a mouse button was pressed. This implies that there may be more than one anchor containing that byte, and all these anchors will be selected at once. In order to find the anchor(s) containing a given byte we need a method to find out whether a byte precedes or follows another byte in the same node.

Below we show the GOOD query which finds all anchors containing the byte with id 123. Once an anchor is found the query to follow the links of which the anchor is the source is trivial. Answering such a query consists of creating an answer object and adding edges that point to the answer of the query (in this case anchors containing the given byte).

```
begin 123
the answer
end
```

Following a link originating at a given anchor or node is very straightforward. (However, the user-interface probably should provide the user with some kind of menu to select a destination from the many links that may be originating from any of the anchors containing the byte the user pointed to with the mouse.) Also, with a few simple object- and edge-additions one can create a trace (or history) which makes it easy to keep track of where the user has been and also to go back to a node one read before. We will omit the pictorial description of these queries.

Another common action in hypertext systems is searching for the occurrences of a word or regular expression for which no explicit links have been provided. Although most systems offer search capabilities these cannot be incorporated in their model since the model hides the contents of the nodes. In the GOOD reference model one can easily find the set of nodes containing a pattern. The query below returns edges pointing to the nodes containing the character 't' followed by an any number of arbitrary characters and then followed by an 'x' and a 't'. This is usually denoted as the search for the regular expression 't*xt'.
Searching for patterns with negation is a bit more difficult as we can only search for objects and edges that are present in a GOOD instance. However, we do have a construction for detecting the absence of an edge, as was shown in the constraint that every node must have an id. In general however, queries with negation are easier to describe in the declarative language G-Log [9] than in the procedural GOOD query language.

7. Conclusions

We have shown how to describe hypertext databases in the GOOD reference model. A basic model was defined for all kinds of data, and a more specific definition of the internal structure of text-nodes was given to show how different kinds of nodes can easily be defined. By describing the internal structure of nodes the GOOD reference model is able to capture a larger part of a hypertext system in one uniform formalism.

Typical hypertext operations: selecting a button, performing a regular expression (or keyword) search and other browsing and query operations are easily described using the same graphical formalism in which the basic building blocks of a hypertext have been defined. Following a link for instance is not fundamentally different from searching for a string. Both are just database queries in GOOD. For more complicated queries or constraints the procedural GOOD query language can be replaced by the declarative language G-Log.

The use of a single formalism for browsing the static and dynamic information in a hypertext makes the GOOD reference model come closer to the "goals" for future generations of hypertext systems, described in [6], especially those describing the need for more powerful search and query operations and the availability of virtual structures for dealing with changing information. New goals will of course arise in the future. This paper provided evidence that the GOOD reference model is a good model for describing hypertext and that the flexibility and straightforward semantics of the GOOD database model make it very easy to extend in the future.
REFERENCES


**In this series appeared:**

<table>
<thead>
<tr>
<th>Volume</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>89/1</td>
<td>E.Zs.Lepoeter-Molnar</td>
<td>Reconstruction of a 3-D surface from its normal vectors.</td>
</tr>
<tr>
<td>89/2</td>
<td>R.H. Mak, P.Struik</td>
<td>A systolic design for dynamic programming.</td>
</tr>
<tr>
<td>89/3</td>
<td>H.M.M. Ten Eikelder, C. Hemerik</td>
<td>Some category theoretical properties related to a model for a polymorphic lambda-calculus.</td>
</tr>
<tr>
<td>89/4</td>
<td>J.Zwiers, W.P. de Roever</td>
<td>Compositionality and modularity in process specification and design: A trace-state based approach.</td>
</tr>
<tr>
<td>89/5</td>
<td>Wei Chen, T.Verhoeff, J.T.Udding</td>
<td>Networks of Communicating Processes and their (De-)Composition.</td>
</tr>
<tr>
<td>89/6</td>
<td>T.Verhoeff</td>
<td>Characterizations of Delay-Insensitive Communication Protocols.</td>
</tr>
<tr>
<td>89/7</td>
<td>P.Struik</td>
<td>A systematic design of a parallel program for Dirichlet convolution.</td>
</tr>
<tr>
<td>89/9</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: Dynamic versus static topology.</td>
</tr>
<tr>
<td>89/10</td>
<td>S.Ramesh</td>
<td>A new efficient implementation of CSP with output guards.</td>
</tr>
<tr>
<td>89/11</td>
<td>S.Ramesh</td>
<td>Algebraic specification and implementation of infinite processes.</td>
</tr>
<tr>
<td>89/12</td>
<td>A.T.M.Aerts, K.M. van Hee</td>
<td>A concise formal framework for data modeling.</td>
</tr>
<tr>
<td>89/14</td>
<td>H.C.Haesen</td>
<td>ELDA, data manipulatie taal.</td>
</tr>
<tr>
<td>89/15</td>
<td>J.S.C.P. van der Woude</td>
<td>Optimal segmentations.</td>
</tr>
<tr>
<td>89/16</td>
<td>A.T.M.Aerts, K.M. van Hee</td>
<td>Towards a framework for comparing data models.</td>
</tr>
<tr>
<td>89/17</td>
<td>M.J. van Diepen, K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra.</td>
</tr>
</tbody>
</table>
Formal methods and tools for the development of distributed and real time systems, p. 17.

Dynamic process creation in high-level Petri nets, pp. 19.

Foundations of Compositional Program Refinement - safety properties -, p. 38.

Decomposition of delay-insensitive circuits, p. 25.

On the delay-sensitivity of gate networks, p. 23.


A logic for one-pass, one-attributed grammars, p. 14.

Receptive Process Theory, p. 16.

Combining the functional and the relational model, p. 15.

A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).

A proof system for process creation, p. 84.

A proof theory for a sequential version of POOL, p. 110.

Proving termination of Parallel Programs, p. 7.

A proof system for the language POOL, p. 70.

Compositionality in the temporal logic of concurrent systems, p. 17.

A fully abstract model for concurrent logic languages, p. 23.

On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.
Design and implementation aspects of remote procedure calls, p. 15.

Two Case Studies in ExSpect, p. 24.

The Nature of Delay-Insensitive Computing, p. 18.

Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.


Implication, A survey of the different logical analyses "if...then...", p. 26.

Parallel Programs for the Recognition of $P$-invariant Segments, p. 16.

Performance Analysis of VLSI Programs, p. 31.

An Implementation Model for GOOD, p. 18.

SPECIFICATIEMETHODEN, een overzicht, p. 20.

CPO-models for second order lambda calculus with recursive types and subtyping, p.

Terminology and Paradigms for Fault Tolerance, p. 25.

Interval Timed Petri Nets and their analysis, p. 53.

POLYNOMIAL RELATORS, p. 52.

Relational Catamorphism, p. 31.


A note on Extensionality, p. 21.

The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15 A.T.M. Aerts
   K.M. van Hee

91/16 A.J.J.M. Marcelis
   An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17 A.T.M. Aerts
   P.M.E. de Bra
   K.M. van Hee
   Transforming Functional Database Schemes to Relational Representations, p. 21.

91/18 Rik van Geldrop
   Transformational Query Solving, p. 35.

91/19 Erik Poll
   Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.

91/20 A.E. Eiben
   R.V. Schuwer

91/21 J. Coenen
   W.-P. de Roever
   J. Zwiers
   Assertional Data Reification Proofs: Survey and Perspective, p. 18.

91/22 G. Wolf

91/23 K.M. van Hee
   L.J. Somers
   M. Voorhoeve
   Z and high level Petri nets, p. 16.

91/24 A.T.M. Aerts
   D. de Reus
   Formal semantics for BRM with examples, p.

91/25 P. Zhou
   J. Hooman
   R. Kuiper
   A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.

91/26 P. de Bra
   G.J. Houben
   J. Paredaens
   The GOOD based hypertext reference model, p. 12.

91/27 F. de Boer
   C. Palamidessi
   Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.

91/28 F. de Boer
   A compositional proof system for dynamic process creation, p. 24.