A Set Theory Based Approach on Applying Domain Semantics to XML-Structures

Wolfgang Schuetzelhofer
IBM Austria
Obere Donaustrasse 95
A-1020 Vienna, Austria
wschuet@attglobal.net

Karl M. Goeschka
Vienna University of Technology
Gusshausstrasse 27-29/384
A-1040 Vienna, Austria
goeschka@acm.org

Abstract

As XML is gathering more and more importance in the field of data interchange in distributed business to business (B2B) applications, it is increasingly important to provide a formal definition of XML-structures together with a well defined way to map business domain semantics to these structures. An XML-algebra, similar to the relational algebra, is required for the formal definition of operations and transformations and to prove the correctness and completeness of design methods.

To develop an XML-algebra, we propose a sound mathematical foundation, modeling XML-structures as typed directed graphs based on set theory. Together with a formal method to apply domain semantics to directed graphs we present a three layer meta model to address the separation of structure and content, and we introduce extensible type hierarchies on nodes and links. This allows to model and validate business domain semantics on different levels of abstraction.

1 Problem statement and goal

The importance of modeling business domain semantics, thus focusing on the user's perception of data, in contrast to physical data representation, has been highly recognized for many years. The field of research on semantic data models has grown rapidly over the last years [2, 5, 10, 11, 13].

With XML as a standardized and broadly used data description language, this area is even gathering more importance. As more and more XML schemas are introduced for different domains, it is increasingly important to provide a formal definition of XML-structures together with a well defined way of mapping domain semantics to these structures. As standards for XML-based B2B infrastructures (ebXML, UDDI) are evolving rapidly, new requirements for a sound mathematical foundation arise.

To ensure robustness, XML-data which serve as input for business applications, have to be proved for validity with respect to schemas. Algebraic specifications, based on algebraic equations, are a formal means to define validity.

XML-based data exchange between different systems requires transformations of XML-structures. These transformations in turn have to be proved for equivalence and soundness. Transformations are also needed as building blocks for design methods in order to develop XML-based data models for different business domains. On one hand this requires a formal method to map business domain semantics to XML-structures. On the other hand it requires an appropriate algebra, similar to the relational algebra for RDBMS, in order to prove the correctness and completeness of single transformations and the whole design method respectively.

As more and more business domains are modeled using XML, querying XML-data becomes another focal point. Querying structured data requires the formal definition of operations, mappings and transformations.

Therefore, similar to RDBMS, where the relational algebra serves as a basis for design methods and formal proofs, an appropriate algebra is required for XML to allow formal specification and verification. As a first step a sound mathematical foundation has to be build. Towards such an algebra, the contribution of our paper is:

- to provide a sound, set theory based foundation to model XML-structures,
- to develop a mathematical formalism in order to express validity of XML-structures,
- to provide a formal method how to apply domain semantics to XML-structures,
- to outline, how the elaborated results satisfy the requirements of building algebraic specifications and an XML-algebra, and how this work contributes to the field of formal software engineering.

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

2 Key concepts

In this section we introduce the key concepts which are elaborated in more detail in the remainder of this paper.

Modeling XML-structures as directed graphs

XML models documents and data structures as a composition of elements. The basic structure of an XML-document is given by the hierarchically nested elements, thus it is a natural approach to model an XML-document as a tree-graph, where the edges of the graph reflect the parent – child relationship of elements.

This is the approach of the World Wide Web Consortium (W3C) in its specification of the Document Object Model called DOM. DOM defines a set of interfaces, using OMG's CORBA IDL [14], to model an XML-document as an in-memory tree of objects.

Several linking mechanisms such as ID/IDREF or the XML Linking Language [15] extend the basic tree structure of XML-documents to express more complex relationships between XML-elements.

In order to provide a seamless integration of different linking mechanisms, our concept is to model XML-structures as directed graphs rather than as tree-graphs.

Types as a means to map business domain concepts

Applying business domain concepts to XML-structures is a key issue of this work. Research work in the field of semantic nets is providing some ideas how to approach this issue. Hypertext systems, which in fact are the origin of XML, may profitably be viewed as semantic nets [1,7]. A semantic net is a directed graph in which concepts are represented as nodes and relations between concepts are represented as links. Using this approach, we introduce types on nodes and links of a directed graph and we appropriately map business domain concepts and relationships between business domain concepts to node and link types.

Type hierarchies as a means of abstraction

The process of modeling business domains requires the possibility to view and describe domain models at different levels of abstraction. To provide this ability, we introduce hierarchies on node types and link types.

Expressing business domain knowledge through structural constraints

Business domain concepts and relationships between business domain concepts are mapped to node types and link types. Our concept to express business domain knowledge and rules is to define structural constraints on these types.

A three-layer meta model

The consequent separation of structure and content, together with specifying type dependent structural constraints, enforces the introduction of a three-layer meta model.

3 Defining a typed graph based on set theory

This section introduces a definition of a directed graph with typed nodes and typed links based on set theory. The graph is called a typed graph and is denoted **TG**.

3.1 Directed graphs and link composition

A directed graph **DG** consists of a set of vertices (nodes) and of a set of edges, which connect the nodes:

DG = (N , E) with **N** a finite nonempty set of nodes

N = {n_1,n_2, ... ,n_n} and

E Í **N x N**

a subset of the Cartesian product **N x N** containing edges of the graph.

A directed graph only has directed edges. A directed edge connects a start node (or source node) to an end node (or target node). A path through a graph is defined as a connected finite sequence of edges.

We introduce the term *link* to abstractly express a connection between two nodes of a directed graph. In the common case such a link represents a path through the graph (in contrast to other publications, where *links* are used as synonyms for *edges*). We define a link in terms of its start node, its end node and a possibly empty sequence of connected intermediate links. In case of an empty sequence of intermediate links, such a link represents an edge of the graph and is called *direct link*.

This allows to recursively compose a link out of other links, with *direct links* representing the non recursive exit. Contrasting the common two-level approach, which distinguishes between edges and paths, our approach of link composition is more powerful and serves as an important means of abstraction.

We redefine a directed graph to be expressed in terms of nodes and links:

DG = (N , L) where

L Í **N_S x N_E x LS**

is a set of links **l** with **N_S** Í **N** a set of start nodes and **N_E** Í **N** a set of end nodes and **LS** a set of connected link sequences. A set of connected link sequences is defined:

LS Í P $_{n \in \underline{N}}$ **L_n** so that

LS = {(l_1, l_2, ..., l_k) | k ³ 0 Ù " i, 1 £ i £ (k – 1) : t(l_i) = s(l_{i+1})}

For every two succeeding links in the sequence, the end node of the first link is the start node of the second link.

s(l), **t(l)** are two functions defined on a link **l**, so that:

s(l) = n_s ; t(l) = n_e ... **s** maps a link to its start node **n_s** and **t** maps a link to its end node **n_e**.

A connected link sequence (element of **LS**) is denoted **ls**, and **ls^0** is the special case of an empty link sequence.

We can complete the recursive definition of links:

L Í **N_S x N_E x LS**

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

L = {á n_s, n_e, lsñ | ls = ls^0 Ú " ls^1 ls^0 : s(ls) = n_s Ù t(ls) = n_e }

where **s(ls)**, **t(ls)** are two functions on a link sequence **ls**, so that **s(ls)** maps a link sequence to its start node and **t(ls)** maps a link sequence to its end node.

Whenever we need to distinguish between a link representing a path through the graph and a link representing an edge of the graph, we denote the latter one as l^0, reflecting the fact that it is the special case of a link án_s, n_e, ls^0ñ. A link l^0 is called *direct link*.

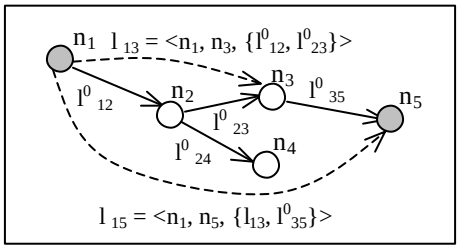

Figure 1 – Recursive link composition

3.2 Introducing node types and link types

We have stated our goal to apply business domain semantics to XML-structures. We have motivated that modeling XML-structures as directed graphs is a profitable approach in order to seamlessly integrate different linking mechanisms. Interpreting XML itself as a business domain requires a mechanism to map business domains to directed graphs. The approach we propose is to introduce types on nodes and links of a directed graph, so that concepts of the business domain are mapped to node types and relationships between business domain concepts are mapped to link types.

We introduce a set of node types **NT** and a set of link types **LT**. Elements of **NT** are denoted t_{node} and elements of **LT** are denoted t_{link}. We introduce the notion of a typed node **tn** and we introduce the notion of a typed link **tl**. The set of typed nodes written **TN** is specified as a mapping of nodes to node types **TN : N ® NT**, and the set of typed links **TL** is specified as a mapping of links to link types **TL : L ® LT**.

Elements of **TN** are denoted

tn = á n, t_{node}ñ n Î N Ù t_{node} Î NT

Elements of **TL** are denoted

tl = á tn_i, tn_j, tls , t_{link}ñ tn_i, tn_j Î TN Ù t_{link} Î LT

A typed link connects a typed node as start node to another typed node as end node through a possibly empty sequence of intermediate typed links.

A typed element (typed node or typed link) is called *instance* of its type.

Additionally we define a function **value(tn)** on typed nodes, which maps a typed node to its node value **n**.

value(á n, t_{node}ñ) = n

and a function **type(tn)** on typed nodes, which maps a typed node to its node type t_{node}.

type(á n, t_{node}ñ) = t_{node}

A function **type(tl)** maps a typed link to its link type

type(á tn_i, tn_j, tls , t_{link}ñ) = t_{link}

Next we state that the value of a typed node is out of a set of possible values which we call a domain. The domain (set of possible values of a typed node) is specified on a per type basis and is denoted:

dom(type(tn)) ... the domain of a typed node, so that

if **tn = á n, t_{node}ñ** is a typed node

then **n Î dom(type(tn))** or **n Î dom(t_{node})**

We can formally express:

value(tn) Î dom(type(tn))

The tuple **GT = (NT, LT)** is called **G**raph **T**ype.

We introduce the notion of a **T**yped **G**raph **TG** as

TG = (TN , TL, dom(TG))

With the domain of the typed graph denoted:

dom(TG) = È$_{tn \hat{I} TN}$ dom(type(tn))

TG consists of three elements: a set of typed nodes, a set of typed links and a set **dom(TG)** being called the domain of the typed graph **TG**.

4 Developing a three-layer meta model

In the previous section we have introduced node types and link types as a means to map business domain concepts and relationships between business domain concepts to typed graphs.

The additional ability to express business domain knowledge is crucial for the quality of a business model. Our approach to express business domain knowledge is to define constraints on types of a **TG**.

In this section we furthermore show how the consequent separation of type and instance, with the goal to separate structure and content, leads to a three-layer meta model approach. And we prove, how structural constraints, covering business domain knowledge, seamlessly integrate into this model.

4.1 Separating type and instance

We define two levels on which to specify elements, sets and relations. One is called *type-level* and the other one is called *instance-level*. Elements defined on the type-level are called types, which in our case are link types, node types and graph types. Elements defined on the instance-level are called instances (or typed elements) and in our case are typed nodes, typed links and typed graphs.

Due to the property of mappings, many instances can exist for one type, but one instance must be of one and only one type (we use the term *instance-of*). Figure 2 pictures the two-level approach.

In the following sections we sometimes use the term link instead of typed link and the term node instead of typed node when the meaning is clear. We do this in order to simplify the formulation of certain statements.

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

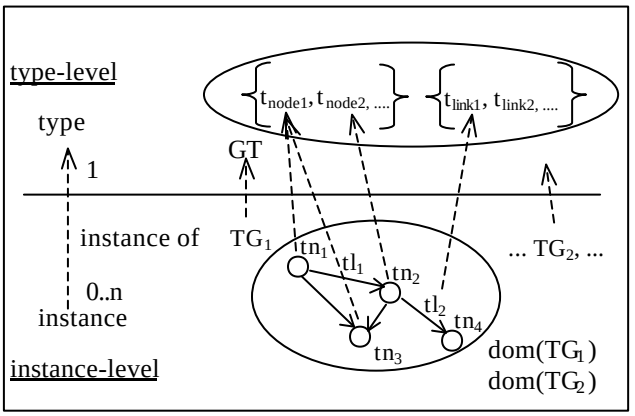

Figure 2 – Separation of type and instance

4.2 Specifying type hierarchies

The means to map business domains to directed graphs are types (business domain concepts are mapped to node types, relationships between business domain concepts are mapped to link types).

The process of modeling business domains requires the possibility to describe and view those domains at different levels of abstraction. To provide different levels of abstraction we introduce type hierarchies.

We define a directed graph, called a type hierarchy, which is denoted $\mathbf{T_H}$.

$\mathbf{T_H = (T , L_T)}$ ... where **T** is a set of types and $\mathbf{L_T}$ is a set of *direct links* (edges), specifying relations between types.

$\mathbf{L_T = \{ t_i, t_j, l_Ts \mid t_i, t_j \in T \wedge t_i \neq t_j \wedge l_Ts = l_Ts^0 \}}$

In order to make $\mathbf{T_H}$ a hierarchy graph, we have to apply the following constraints:

$\mathbf{\forall G^-(t_i) \leq 0..1}$ where

$\mathbf{G^-(t_i) = \{ m \mid \exists m , t_i, l_Ts^0 \in L_T \}}$

which formally defines that a type can have at most one super-type. A type having no super-type is called root-type. In addition we define:

$\mathbf{\forall t \in T : t \notin D(t)}$ where

$\mathbf{D(t) = \{ m \mid \exists l_Ts : s(l_Ts) = t \wedge t(l_Ts) = m \}}$ ... $\mathbf{l_Ts}$ is a connected sequence of links $\mathbf{l_T \in L_T}$. **D(t)** is the set of descendants of **t** and hence denotes the set of all sub-types of **t**. This means the type hierarchy graph is acyclic.

A type hierarchy allows type dependent characteristics and structural constraints being described, viewed and handled at different levels of abstraction through the means of specialization and generalization.

On the type-level a type is said to be a specialization of its super-type, and a type is said to be a generalization of its sub-type. On the instance-level, in any place where an instance of a specific type can be used, an instance of any of this type's sub-types can be used as well. This, in a similar form, is known as polymorphism in the domain of object oriented development.

This concept has some important implications. On the type-level, a type specifies characteristics and structural constraints to be applied on instances of that type. Instances (typed elements) have to satisfy not only the constraints defined by their specific type but also the constraints defined by all that type's super-types in the type hierarchy. Or the other way round, constraints and characteristics specified by a certain type have to be applied on instances of that type as well as on instances of all sub-types of that specific type.

For a graph type we extend the definition of node types and link types to be type hierarchies

$\mathbf{NT_H = (NT , L_{NT})}$ ...node type hierarchy
$\mathbf{LT_H = (LT , L_{LT})}$ ...link type hierarchy,
$\mathbf{GT = (NT_H, LT_H)}$ ...graph type

4.3 Introducing structural constraints

Structural constraints are the means to describe business knowledge and business rules for a certain domain. A sales transaction requiring one customer, at least one sales item and exactly one billing address to be defined, is an example of such business knowledge. Business domain concepts and relationships between these concepts are mapped to node types and link types. We say that a graph type defines the structure of a business domain, and consequently structural constraints have to be specified on types and hence to be defined at the *type-level*. Our approach is to model structural constraints as nodes and links of a directed graph at the *type-level*.

In section 3.2 we have specified two hierarchy graphs at the *type-level*, one for node types and one for link types. We now extend the definition of the graph type **GT** to be a directed graph at the *type-level*, so that the node type hierarchy as well as the link type hierarchy are partial graphs of **GT**. Node types and link types are nodes of **GT**. Every structural constraint in turn is represented as a node of **GT**. A node type or link type (node of GT) which is structurally constraint, is linked to the node representing the constraint.

Consequently going one step further we define **GT** to be a typed graph with link types, node types and with constraints being *typed* nodes of this graph.

4.4 A three-layer meta model

The last step, described in the previous section, leads to the introduction of a three-layer meta model.

We have defined a typed graph in terms of typed nodes and typed links. We said typed nodes and typed links are called *instances of* the appropriate node and link types. In the two-level approach we introduced types (node types, link types) to be specified at the type-level.

Now we say that node types and link types themselves are typed nodes of a typed graph at the type-level. Consequently a node type or link type itself must be an *instance of* ... but what ?

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

The solution to this problem is the introduction of a third level which we call *meta-type-level*. We introduce this level, so that node types and link types, as typed nodes of the *type-level*, are *instances of meta-types*. With *meta-types* being defined at the *meta-type-level*.

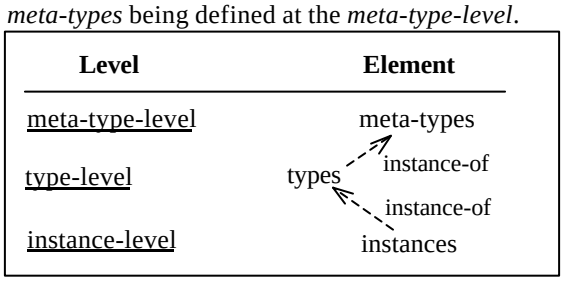

Figure 3 – Three-layer meta model

More generally stated, typed elements at the *instance-level* are *instances of* typed nodes at the *type-level* which in turn are *instances of* node types at the *meta-type-level*.
We can formally express:
TG = (TN , TL, dom(TG))
...typed graph(instance-level)
GT = (GTN, GTL, dom(GT))
...graph type (type-level)
MT = (MNT, MLT)...meta type (meta-type-level)
so that at the *instance-level*:
" **tn** Î **TN : type(tn)** Î **GTN**
" **tl** Î **TL : type(tl)** Î **GTN**
at the *type-level*:
" **gtn** Î **GTN : type(gtn)** Î **MNT**
" **gtl** Î **GTL : type(gtl)** Î **MLT**
Node type hierarchies and link type hierarchies (section 3.2) are partial graphs of **GT** and mapped to **GT** so that node types as well as link types are nodes of **GT**, and node type- and link type relations are links of **GT**.

With **MNT** a set of node types at the *meta-type-level*, elements of **GTN** are defined:
gtn = á**nt, mt$_{node}$**ñ with
nt Î **dom(mt$_{node}$)** Ù **mt$_{node}$** Î **MNT**
and with **MLT** a set of link types defined on the *meta-type-level*, elements of **GTL** are defined:
gtl = á **gtn$_i$, gtn$_j$, gtls , mt$_{link}$**ñ **gtn$_i$, gtn$_j$** Î **GTN**
Ù **mt$_{link}$** Î **MLT**
and **gtls** a possibly empty sequence of intermediate links (elements of **GTL**).

5 The domain graph model

The three layer meta model of a typed graph, including a formal method to map business domains, in terms of domain concepts and business knowledge (-rules), is called **Do**main **G**raph (**DoG**).

To illustrate how a **DoG** is specified, how validity of a **DoG** is expressed and how the three levels (*instance-level*, *type-level*, *meta-type-level*) are related, we introduce an example of a simple business domain. We model persons identified by their name, which are each associated with a telephone identified by a telephone number. To be simple we model a single person named 'Smith' being associated to a telephone with number '12345'. By means of this example we will first demonstrate how different types of constraints (expressing business knowledge) are specified.
The constraints to be covered are:

- Restricting start node type and end node type of a link
- Constraining link cardinalities
- Constraining intermediate link sequences in terms of intermediate link types

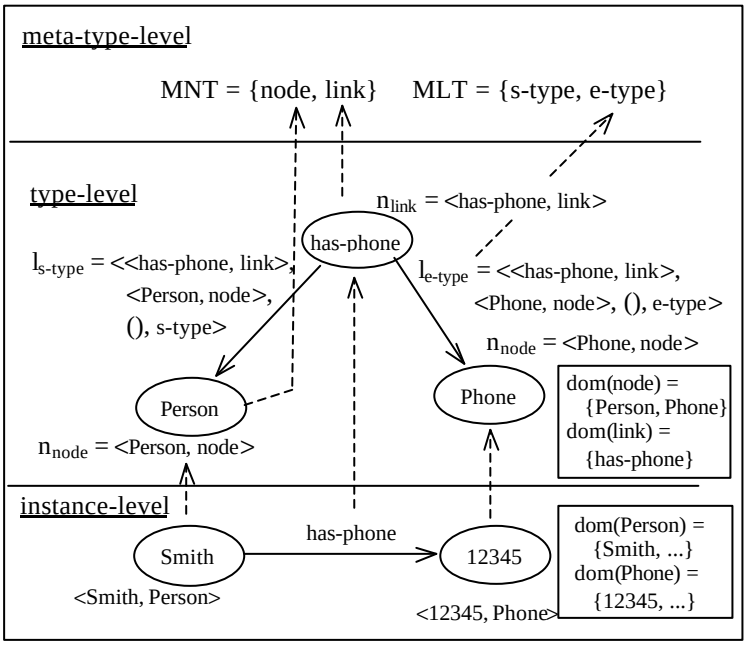

Figure 4 – Example model

At the *type-level*, outlined in figure 4, the business domain is modeled as follows:

- Business domain concepts are represented as nodes of *meta-type* **node**. In our case these are: **Person** and **Phone**.
- Relationships between business domain concepts are represented as nodes of *meta-type* **link**. In our case: **has-phone**.
- Structural constraints are represented as nodes and / or links at the *type-level*.
- The semantic of a constraint is specified by the *meta-type* of the constraint link. Meta-type **s-type** constrains the start node type of a link, meta-type **e-type** constrains the end node type of a link. In our case the start node type of **has-phone** is **Person**, the end node type of **has-phone** is **Phone**.

To introduce cardinality constraints we say that a person is allowed to have up to 5 telephones, but a telephone can only be owned by a single person. This is

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

drawn in figure 5 (only the relevant fragment of the example is drawn) .We define a cardinality constraint as a node on the *type-level* with *meta-type* **card**. The constraint is connected to the node which represents the link type to be constraint.

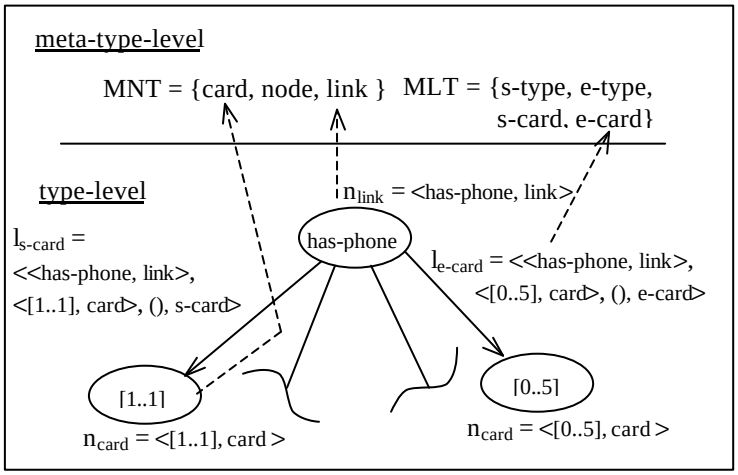

Figure 5 – Modelling cardinality constraints

The range of possible values (domain) for cardinality constraints is defined based on the meta-type.

dom(card) Í ***N*** **x** ***N***

dom(card) = {(min, max) | min, max Î ***N*** Ù **min** £ **max}**

The semantic of the constraint is specified by the *meta-type* of the constraint link, which can be **s-card** and **e-card**.

s-card constrains the number of instances of a link type which connect one instance of the start node type to instances of the end node type to be within the range defined by min, max of the cardinality constraint inclusively.

e-card constrains the number of instances of a link type which connect instances of the start node type to one instance of the end node type to be within the range defined by min, max of the cardinality constraint inclusively.

A sequence of link types (nodes of *meta-type* **link**) constrains the sequence of intermediate link instances of an indirect link. We call it a *sequence constraint* and introduce *meta-node-type* **sequ** Î **MNT**.

We extend our example so that a phone is *provided-by* a telephone company. Hence we need to model the association between phone and company. Additionally we define an indirect link, reflecting that a person having a phone which in turn is provided by a telephone company is a customer of that telephone company (see figure 6).

Note that the link *customer-of* in figure 6 is drawn as a dotted line in contrast to other (direct) links, which are drawn as solid lines.

We specify the domain of possible values for a *sequence constraint*. We first define two functions **s-node**, **e-node** on nodes $\mathbf{gtn_{link}}$ at the *type-level*. The set $\mathbf{GTN_{link}}$ of elements $\mathbf{gtn_{link}}$ is defined as:

$\mathbf{GTN_{link}}$ Í **GTN**

$\mathbf{GTN_{link}}$ **= {gtn | type(gtn) = link}**

A node $\mathbf{gtn_{link}}$ represents the type of a link.

The function **s-node** maps $\mathbf{gtn_{link}}$ to the node $\mathbf{gtn_s}$ which specifies the start node type of a link.

With $\mathbf{type(l_{s\text{-}type}) = s\text{-}type}$:

" $\mathbf{gtn_{link}, gtn_s}$ Î **GTN "** $\mathbf{l_{s\text{-}type}}$ Î **GTL :**

$\mathbf{[s\text{-}node(gtn_{link}) = gtn_s}$ Þ **$** $\mathbf{l_{s\text{-}type}}$ **: s($\mathbf{l_{s\text{-}type}}$) = $\mathbf{gtn_{link}}$** Ù **t($\mathbf{l_{s\text{-}type}}$) = $\mathbf{gtn_s}$]**

In quite the same way the function **e-node**, which maps a node $\mathbf{gtn_{link}}$ to the end node type $\mathbf{gtn_e}$, is formally expressed.

The domain dom(sequ) is defined:

dom(sequ) Í $\mathbf{P_{n\ \hat{I}\ \underline{N}}\ GTN_{link\ n}}$

dom(sequ) = {($\mathbf{gtn_{link\ 1}}$, $\mathbf{gtn_{link\ 2}}$, ..., $\mathbf{gtn_{link\ k}}$) |

k ³ **1** Ù **" i, 1** £ **i** £ **(k – 1) : e-node($\mathbf{gtn_{link\ i}}$) = s-node($\mathbf{gtn_{link\ i+1}}$)}**

An empty sequence constrains a link to be an edge of the graph. Additionally we define subdomains denoted:

dom(sequ, $\mathbf{gtn_{link}}$) so that

dom(sequ, $\mathbf{gtn_{link}}$) Í **dom(sequ)**

dom(sequ, $\mathbf{gtn_{link}}$) = {($\mathbf{gtn_{link\ 1}}$, $\mathbf{gtn_{link\ 2}}$, ..., $\mathbf{gtn_{link\ k}}$) |

k ³ **1 : s-node($\mathbf{gtn_{link\ 1}}$) = s-node($\mathbf{gtn_{link}}$)** Ù **e-node($\mathbf{gtn_{link\ k}}$) = e-node($\mathbf{gtn_{link}}$)}**

A subdomain is specific for a certain node $\mathbf{gtn_{link}}$ and contains only link type sequences which start at the start node of $\mathbf{gtn_{link}}$ and which end at the end node of $\mathbf{gtn_{link}}$.

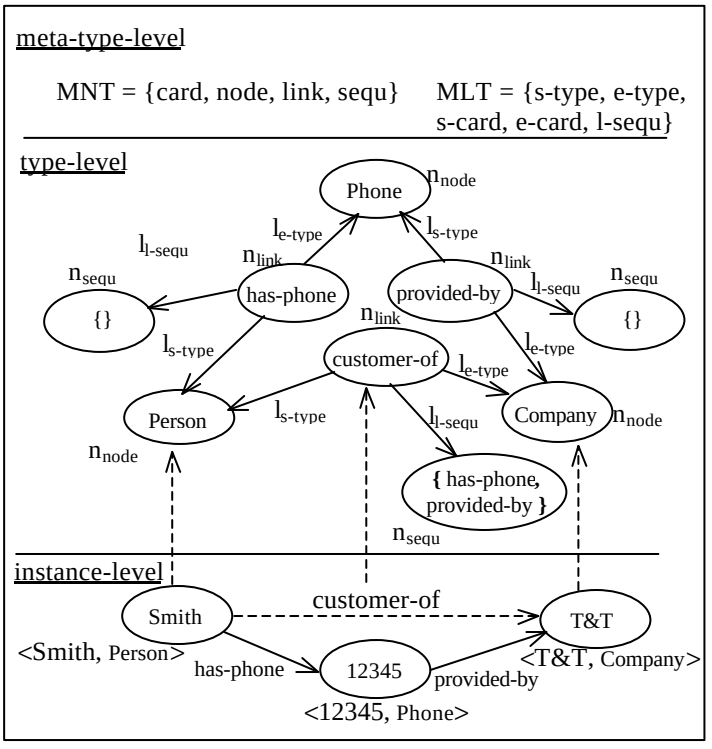

Figure 6 – Indirect links and sequence constraints

At this point we want to give a motivation on introducing link composition as a powerful means of abstraction. A typed graph is in first order defined by its nodes and by its direct typed-links $\mathbf{tl^0}$, whereas indirect links are implicit to the graph. Indirect links are derived by navigating along connected link-sequences through the graph, thus revealing implicit information covered by the graph. This implicit information is made explicit by mapping such link sequences to distinct links of specific type. This on one hand provides for extracting relevant, domain specific information out of a complex graph and on the other hand allows to raise a complex graph to a more abstract level, focusing on domain concepts and relationships between domain concepts of interest.

5.1 Validity of a typed graph

We introduce the notion of *validity* of a typed graph **TG**: A typed graph is *valid* if and only if it satisfies the structural constraints defined by its graph type.

Formalizing validation is an important issue, as it builds the basis for developing an algebraic specification, which in turn provides for proving the soundness and completeness of transformations defined on typed graphs.

In section 3.2 we have outlined the advantages and the implications of introducing type hierarchies. To take these implications into account, we introduce some more formalisms on type hierarchies.

On type hierarchy graphs we define sets called **Sub-Types** and sets called **Super-Types**. The sets contain sub-types and super-types of a given element **gtn** (typed node at the *type-level*). Depending on **type(gtn)** which can either be **node** or **link** (specified at the *meta-type-level*), we are looking at the node type hierarchy or at the link type hierarchy. We additionally specify **sub-type** Î **MLT** to represent a type hierarchy link. Now we define:

$\mathbf{Sub\text{-}Types_{gtn} = \{gtn_i \mid type(gtn) = (node \acute{U} link) \grave{U} type(gtn_i) = type(gtn) \grave{U} \$\ gtls :}$
$\mathbf{[s(gtls) = gtn \grave{U} t(gtls) = gtn_i \grave{U} "\ gtl \hat{I} gtls :}$
$\mathbf{type(gtl) = sub\text{-}type]\}}$

... **gtls** is a connected sequence of links **gtl** Î **GTL**.

$\mathbf{Sub\text{-}Types^{+}_{gtn} = \{gtn\} \grave{E} Sub\text{-}Types_{gtn}}$

$\mathbf{Super\text{-}Types_{gtn} = \{gtn_i \mid gtn \hat{I} Sub\text{-}Types_{gtn\,i}\}}$

$\mathbf{Super\text{-}Types^{+}_{gtn} = \{gtn\} \acute{E} Super\text{-}Types_{gtn}}$

We have already defined the function **type()** on typed elements, in addition we now define a function **instance-of()** on typed elements. The function **instance-of()** maps a tuple consisting of a typed element and a type to a boolean value out of the set **{true, false}**, so that with **TN** a set of typed nodes, **GTN** a set of types:

instance-of : (TN x GTN) ® {true, false} written:
instance-of(tn, gtn) Î {true, false}, tn Î TN Ù gtn Î GTN the mapping is defined as:

" tn Î TN " gtn Î GTN : [instance-of(tn, gtn) = true Þ type(tn) Î Super-Types$^{+}_{gtn}$]

In order to formalize validation, we define the function **valid** on a typed graph. The function **valid** maps a typed graph to the set of boolean values **{true, false}**.

valid : TG ® {true, false} we write:
valid(TG) Î {true, false}

In case **valid(TG) = true**, the typed graph is valid, else it is invalid. V*alidity* of a typed graph is defined in terms of *validity* of the graph's nodes and of the graph's links so that:

TG = (TN , TL, dom(TG)) :
[valid(TG) = true Þ " tn Î TN : valid(tn) = true Ù " tl Î TL : valid(tl) = true]

A typed graph is valid if and only if all typed nodes and all typed links of the graph are valid. We define additional functions to express validity of a single node and of a single link with respect to the specified constraints.

For a typed node **tn**, on its associated type $\mathbf{t_{node}}$ = **type(tn)**, we select a set of outgoing link types $\mathbf{t_{link}}$. With $\mathbf{t_{node}}$ Î **GTN**:

$\mathbf{GTN_{tnode,\,out} \acute{I} GTN}$ and
$\mathbf{GTN_{tnode,\,out} = \{ t_{link} \mid s\text{-}node(t_{link}) \hat{I} Super\text{-}Types^{+}(t_{node}) \}}$

The set $\mathbf{GTN_{tnode,\,out}}$ contains link types with the given node type $\mathbf{t_{node}}$ or one of its super-types specified as start node type. This considers the concept of polymorphism.

Similarly we define a set of incoming link types:

$\mathbf{GTN_{tnode,\,in} \acute{I} GTN}$ and
$\mathbf{GTN_{tnode,\,in} = \{ t_{link} \mid e\text{-}node(t_{link}) \hat{I} Super\text{-}Types^{+}(t_{node}) \}}$

We build subsets of typed links containing outgoing links of the typed node **tn** which are instance-of certain link type $\mathbf{t_{link}}$ Î $\mathbf{GTN_{tnode,\,out}}$ We write:

$\mathbf{TL_{tn,\,tlink,\,out} \acute{I} TL}$

$\mathbf{TL_{tn,\,tlink,\,out} = \{tl \mid s(tl) = tn \grave{U} instance\text{-}of(tl, t_{link}) \}}$

with $\mathbf{t_{link}}$ Î $\mathbf{GTN_{tnode,\,out}}$ **Ù** $\mathbf{t_{node}}$ **= type(tn)**

And we define subsets containing incoming links of **tn** which are instance-of a certain link type:

$\mathbf{TL_{tn,\,tlink,\,in} \acute{I} TL}$

$\mathbf{TL_{tn,\,tlink,\,in} = \{tl \mid e(tl) = tn \grave{U} instance\text{-}of(tl, t_{link})\}}$

with $\mathbf{t_{link}}$ Î $\mathbf{GTN_{tnode,\,in}}$ **Ù** $\mathbf{t_{node}}$ **= type(tn)**

As a typed link is instance of its link type but also is instance of any super-type of this link type, such a typed link can be contained in more than one set $\mathbf{TL_{tn,tlink}}$, namely in any set specified on behalf of a super-type of $\mathbf{t_{link}}$. This assures that a typed link (instance-of a link type) is validated against the constraints specified by its link type as well as against the constraints specified by all super-types of its link type. Next we formally define a function **valid-card(tn)** which maps a typed node to **true** or **false**:

tn Î TN, $\mathbf{t_{node}}$ **= type(tn), "** $\mathbf{t_{link\,o}}$ Î $\mathbf{GTN_{tnode,\,out}}$ **,**
" $\mathbf{t_{link\,i}}$ Î $\mathbf{GTN_{tnode,\,in}}$ **:**
[valid-card(tn) = true Þ
÷ $\mathbf{TL_{tn,\,tlink\,o,\,out}}$ **÷** ^{3} **min(s-card(**$\mathbf{t_{link\,o}}$**))** **Ù**

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

÷ $\mathbf{TL_{tn,\ tlink\ o,\ out}}$ ÷ £ **max(s-card($t_{link\ o}$))** Ù
÷ $\mathbf{TL_{tn,\ tlink\ i,\ in}}$ ÷ ³ **min(e-card($t_{link\ i}$))** Ù
÷ $\mathbf{TL_{tn,\ tlink\ i,\ in}}$ ÷ £ **max(e-card($t_{link\ i}$))]**

A typed node is said to be *valid* if and only if all cardinality constraints, specified for outgoing and incoming links, considering the concept of polymorphism, are satisfied.

Now we can formally define how the function **valid(tn)** for elements **tn** Î **TN** and the function **valid(tl)** for elements **tl** Î **TL** map a typed node or a typed link to **true** or **false** respectively.

" **tn** Î **TN : [valid(tn) = true** Þ **valid-card(tn)]**
" **tl** Î **TL : [valid(tl) = true** Þ
instance-of(s(tl) , s-node(type(tl))) Ù
instance-of(t(tl) , e-node(type(tl)))]

In a similar way validity of a graph type can be expressed with respect to constraints defined at the *meta-type-level*.

5.2 Formal specification of a DoG

In the **DoG** model, business domain semantics are defined at the *type-level* by mapping business domain concepts and relationships between business domain concepts to node types and link types. Business domain knowledge is covered by appropriately specifying structural constraints at the *type-level*. Additionally one can define type hierarchies in order to describe domain models at different levels of abstraction.

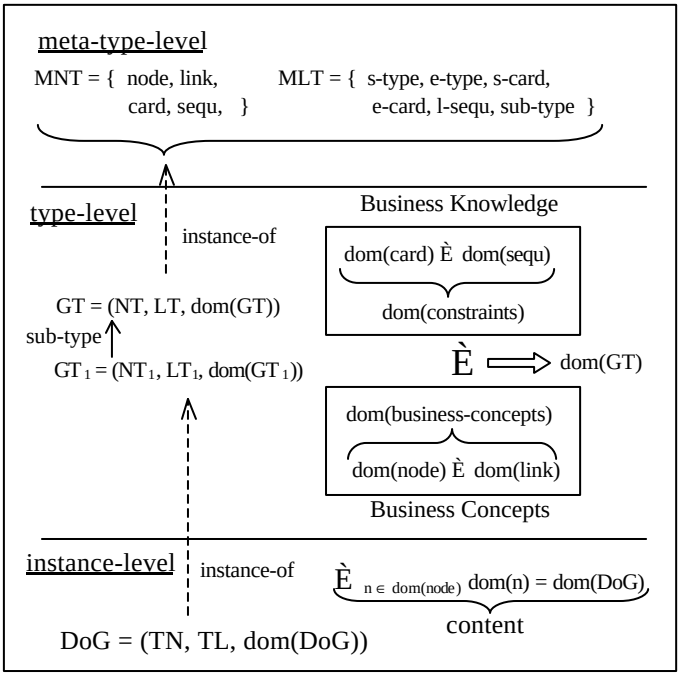

Figure 7 – Mapping business domains to DoGs

To demonstrate how to specify a concrete **DoG**, we refer to our example. We start with specifying domains for node types and link types.

dom(node) = {Person, Phone, Company}
... business domain concepts
dom(link) = {has-phone, provided-by, customer-of}
... relationships between concepts

To specify structural constraints we define a set for every type of constraint-link. For elements of every set we define an abbreviated notion:

$Spec_{(type(gtl))}$ = {á value(gtn_1), value(gtn_2)ñ | gtn_1 = s(gtl) Ù gtn_2 = t(gtl)}
with **gtn_1, gtn_2** Î **GTN** Ù **gtl** Î **GTL** Ù **type(gtl)** Î **MLT** Ù **type(gtn_1), type(gtn_2)** Î **MNT**

The structural constraints for our example are:

$Spec_{(s\text{-}type)}$ = { á has-phone, Person ñ, á provided-by, Phone ñ }
$Spec_{(e\text{-}type)}$ = { á has-phone, Phone ñ, á provided-by, Company ñ }
$Spec_{(s\text{-}card)}$ = { á has-phone, [1..1] ñ, á provided-by, [0..n] ñ }
$Spec_{(e\text{-}card)}$ = { á has-phone, [0..5] ñ, á provided-by, [1..1] ñ }
$Spec_{(l\text{-}sequ)}$ = { á customer-of, {has-phone, provided-by} ñ }

5.3 Mapping XML domain semantics to a DoG

In order to apply XML domain semantics to a **DoG** one has to follow the same pattern as described in the previous section. First the XML domain concepts and the relationships between these concepts have to be defined.

XML specifies documents and data structures to be composed of elements, attributes and data sections which contain text data. An element can be connected to other elements which are said to be the element's children, and an element can reference other elements by means of a mechanism called ID/IDREF mechanism. Additionally an element can have some text content and it can have attributes.

The basic XML domain concepts are: *Element*, *Attribute*, *Text*. The XML Linking Language [15] furthermore specifies simple and extended link-elements which reference other elements, called source and target, by means of XPath (XML Path Language) [16] expressions in order to specify links. As the chosen approach is to define a **DoG** as a directed graph, it is possible to seamlessly integrate the constructs of the XML Linking language into an appropriate type definition.

A complete type specification of the XML-domain following the concepts proposed in this paper is straight forward but it would fill some additional pages and hence exceed the scope of the paper. Once the XML domain concepts and appropriate structural constraints are expressed in form of a graph type for **DoG**s, it is possible to formally specify arbitrary XML-structures and to validate them by means of the set theoretical approach proposed by this paper.

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

5.4 Applying domain semantics through sub-typing

In the previous section we have explained how one can specify a graph type which maps XML domain semantics to **DoG**s in order to define XML-structures.

In this section we outline how to use sub-typing in order to apply business domain semantics to XML-structures. We say that a graph type can be specified to be a sub-type of another graph type in that it defines node types and link types which are sub-types of node- and link types of the other graph type. A graph type in order to be a valid sub-type of another graph type may not introduce root node types or root link types. This is formally expressed as sub-type validity of a graph type with respect to its super-type, written

valid_subtype(GT, GT_{super}) Î **{true, false}**, so that:
GT = (GTN, GTL, dom(GT)),
GT_{super} = (GTN_{super}, GTL_{super}, dom(GT_{super})),
" **t** Î **GTN , type(t) = (node** Ú **link) :**
valid_subtype(GT, GT_{super}) = true Þ
$ **l** Î **GTL : type(l) = sub-type** Ù **t(l) = t** Ù
s(l) Î **GTN** È **GTN_{super}]**

We can specify the domains **dom(node)**, **dom(link)** for a graph type modeling a certain business domain, so that the appropriate sets contain business domain concepts and relationships between business domain concepts respectively. A set of sub-type links has to be specified, so that every business domain concept is expressed as a sub-type of an XML domain concept or of another business domain concept, and that every business domain relationship is expressed as a sub-type of an XML domain relationship or of another business domain relationship. Together with the specification of appropriate constraints, this allows to model a business domain as a sub-type of the XML-domain and hence as special *kind-of* XML-structure.

6 Related work

Beside publications on formal models for hypertext [12] and on constraining hypertext structures [8], related research work based on set theory can be found in the areas of semantic data modeling, especially on applying domain semantics to structured hypertext systems and hyperlinked data sets. A lot of similar approaches stresses the need for our work: In Bench-Capon and Dunne [4], a DAG (directed graph) structure and a set of constraints are used to model electronic documents. In contrast to our paper links are not typed and only represent the containment relationship, providing limited possibilities to express domain semantics. The formal hypertext model described in Tochtermann and Dittrich [3] provides some formally defined structural concepts, lacking mechanisms to define more powerful structural and relational constraints. Wang and Rada [1] have developed a semantic data model based on the concept of a semantic net, introducing organizational and relational link types on a DAG, but in contrast to our work do not provide an extensible type system. Abiteboul and Hull [6] in their approach on formalizing a semantic model recognize the importance of types which are used to model object structures (IFO model), but a concept of link type hierarchies, as elaborated in our paper, is missing.

The W3C's working activity on the XML Schema Language [17] introduces a formal XML-based language to define structural constraints on classes of XML-documents in order to apply domain semantics to these documents. This approach has achieved a high acceptance, as it is targeted towards practical implementations, but the specification does not provide a set theory based mathematical formalism. The Unified Modeling Language (UML) specifies semantics on the level of Meta-Models [18, 9], where a formal graphical language is used together with textual descriptions of the appropriate semantics. A similar approach is proposed by the OMG with its MOF (Meta-Object-Facility) specification [19], introducing a four layer meta-data architecture in contrast to the three layer model we are proposing in our paper.

7 Results and contribution

The results of our work, seen to contribute to the field of formal software engineering are:

- A three-layer meta model, which strictly separates structure and content and which has proven to be a powerful means of abstraction to model complex business domains.
- A hierarchically extensible type system, including the definition of link type hierarchies, a topic which is still missing in related work.
- A formal method to map business domain concepts and to express business knowledge in terms of the meta model.
- The formal definition of domain graph validity in form of mappings, which allows a straight forward definition of algebraic equations and hence the development of algebraic specifications for business domains.
- XML has been defined as a specific business domain, so that sub-typing allows arbitrary business domains to be expressed as sub-types of the XML-domain and hence as special kind-of XML-structure.
- A sound mathematical foundation, using set- and graph theory to describe all aspects of the meta model, considered to serve as a basis for algebraic specifications and for an XML-algebra.

0-7695-1435-9/02 $17.00 (c) 2002 IEEE

8 Conclusion and future work

In this paper we have introduced the concept of a Domain Graph (DoG) as a directed graph with typed nodes and links, being based on what is known as a semantic net. We have outlined, how domain semantics can be applied to a DoG by specifying types and structural constraints. Business domain modeling at the type-level, thus separating structure and content, is proposed as a flexible approach of delivering semantic data.

The concept of type hierarchies as well as the approach of link composition have proved to be powerful means of abstraction.

In future work, the concept of composition as described in this paper can profitably be generalized introducing the notion of composite graphs. Where a composite graph is a DoG with nodes representing entire graphs called component graphs, and with links representing relationships between component graphs. Assuming a component graph itself is a composite graph, this allows nested graph composition of arbitrary depth.

Throughout this paper we have presented static aspects of DoGs using set theory to describe their structure. The set theory based model of a DoG, together with the formal specification of validity, build a sound mathematical basis to develop an XML-algebra allowing to define operations on XML-structures, thus describing dynamic aspects.

Manipulating a DoG by inserting, deleting or updating nodes and links, thereby maintaining consistency and validity of the DoG are such dynamic aspects to be described. Set theory as the mathematical basis provides for formal specifications and proofs of correctness and completeness of design methods by proving the equivalence and soundness of transformations. New and extended design methods, which can be formally specified and verified, are seen to be a profitable output of future work based on this paper.

References

[1] W. Wang, R. Rada, “Structured Hypertext with Domain Semantics”; ACM TIS, 16,4, 1998, pp. 372-412

[2] J.L. Schnase, J.J. Leggett, D.L. Hicks, R.L. Szabo, “Semantic data modeling of hypermedia associations”, ACM Trans. Inf. Syst. 11, 1, Jan. 1993, pp. 27-50

[3] K. Tochtermann, G. Dittrich, “Towards a family of formal models for hypermerdia”, HIM’95 Proceedings, 1995, pp. 77–91

[4] T. Bench-Capon, P. Dunne, “Some computational properties of a model for electronic documents”, Electr. Pub. Orig. Dissem. Des. 2, 4, 1989, pp. 231–256.

[5] J. Peckham, F. Mariansky, “Semantic data models”, ACM Comput. Surv. 20,3, 1988, pp. 153-189

[6] S. Abitoul, R. Hull, “IFO: a formal semantic database model”, ACM Trans. Database Syst. 12, 4, 1987

[7] J. Conklin, “Hypertext: An introduction and survey”, Computer 20, 9, 1987, pp. 17-41

[8] B. Chidlovskii, “Using Regular Tree Automata as XML Schemas”, Proceedings of the IEEE Advances in Digital Libraries 2000, 2000

[9] A. Schleicher, B. Westfechtel, “Beyond Stereotyping: Metamodeling Approaches for the UML”, Proceedings of the 34th Hawaii International Conference on System Sciences 2001 (HICSS-34), 2001

[10] H.A. Schmidt, J.R. Swenson, “On the semantics of the relational data models”, Proceedings of the SIGMOD San Jose, Calif., 1975

[11] Abrial, Klimbie, Koffemen, “Data Semantics In Database Management”, Eds. North-Holland, Amsterdam, 1974, pp. 1-59.

[12] D. Lange, “A formal model for hypertext”, Proceedings of the NIST Hypertext Standardisation Workshop. NIST, Gaithersburg, Md., 1990, pp. 145-166

[13] M. Gogolla, U. Hohenstein, “Towards a semantic view of an extended entity-relationship model”, ACM Trans. Database Syst. 16, 3, 1991, pp. 369-416

[14] OMG CORBA-IDL, *Interface Definition Language Specification*, http://www.omg.org

[15] W3C XLink, *XML Linking Language Proposed Recommendation*, http://www.w3c.org/XML/linking, Dec. 2000

[16] W3C XPath, *XML Path Language Recommendation*, http://www.w3c.org/TR/xpath, Nov. 1999

[17] W3C XML Schema, *XML Schema Specification*, http://www.w3c.org/XML/schema, 2001

[18] UML, *Meta-Model Specification (v1.3)* http://www.rational.com/uml

[19] OMG-MOF, *Meta Object Facility Specification (v1.3)*, http://www.omg.org, march 2000

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35’02)
0-7695-1435-9/02 $17.00 © 2002 IEEE