3. Workman's tools

3.1 Information modelling

In this thesis, we present a technical system that solves problems in the world of electronic design. We have chosen to base its design on an information model.

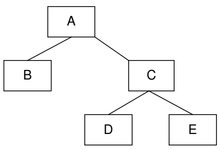
Definition: An *information model* defines the relevant *concepts* in a particular application domain in terms of their *attributes* and *relationships*. The synonym *conceptual model* is often used for information model.

In terms of database technology, an information model abstracts from the fine-grained detail of an actual implementation. To be of any value, the information model must be based on some well-defined formalism.

Definition: A *data model* defines a type system, consisting of a number of base types and constructors to derive new types from existing ones. It also defines a data access and manipulation language that operates on type extents and a language to specify integrity constraints.

Early attempts to define data models stem from the area of database design. The first data models (hierarchical and network data models) directly described the physical data organization supported by a database management system (DBMS) and provided only low-level operations for data manipulations. The relational model is based on the formally defined relational theory and effectively insulates the user of a DBMS from the physical data organization. It provides, however, only scalar datatypes (boolean, integer, real, string) and flat relations as datatypes and therefore is not well suited to model technical domains.

A data model that allows to capture more semantics is the entity-relationship (ER) model by Chen [Chen 76]. This model represents information in terms of entities and the relationships between them. A major advantage of the ER model is that a graphical notation is defined for it which is the usual technique to capture ER models. Since its introduction, many extensions have been proposed to make it more suitable for the modelling of technical systems like electronic circuits. Of these, especially aggregation and generalization are of importance ([Smith 77], [Batory 85]).



type E: INTEGER.
type D: { 'red', 'blue', 'yellow' }.
type C = D, E.
type B: STRING ('default').
type A = B, [C].

Figure 6. *Xplain modelling concepts, graphical notation, and textual schema language. The graphical notation is ordered, i.e. an aggregate is always drawn above its attributes. This way, the edges in a diagram are associated with cardinalities. In the example, there is exactly one element of B for each element of A, while one element of B is related to zero, one, or many elements of A. Furthermore, A is a specialization of C: While C has the attributes D and E, A has the attributes B, D, E.*

With the more widely spread acceptance of the object-oriented concepts and the quest for a richer type system the EXPRESS language, now an ISO proposed standard [Spiby 93], has gained wider acceptance. EXPRESS is an object-oriented language used for information modelling. It comes with a graphical notation (EXPRESS-G) that provides graphical idioms for a subset of the textual EXPRESS language. EXPRESS-G diagrams share a major disadvantage with ER diagrams in that they are unordered. No natural entry point into the diagram can guide the reader in understanding such a diagram. More important, depending on the entry point chosen by a reader, an ER or EXPRESS-G diagram can mean different things.

We have chosen the Xplain semantic data model [terBekke 92] over the other approaches for its few and well-defined concepts and clear graphical notation (Figure 6).

The key concept in Xplain is a type:

Definition: A *type* is defined by a unique name and properties. A type is associated with an *extent*. The extent of a *base type* is a set of either integer or real numbers, strings, or named constants from an enumeration. The extent of an *aggregate type* is a set of objects, each consisting of a unique identifier and a value. The value is a tuple, each element being from the extent of one of the base or aggregate types.

In addition, Xplain allows to *specialize* an aggregate type by adding new attributes to an existing type. For example, Table 7 shows the type extents of the types shown in Figure 6.

type	extent
A	$\{ (b, d, e) \mid b \in B, d \in D, e \in E \}$
B	$\{ e \in Strings \}$
C	$\{ (d,e) \mid d \in D, e \in E \}$
D	{'red', 'blue', 'yellow'}
E	$\{ e \in RealNumbers \}$

Table 7. *Type extents of the example in Figure 6*

Xplain defines two inherent integrity rules for the types in an information model [terBekke 93]:

- *Relatability*
  Each attribute in a textual type definition is related to exactly one type with the same name as the attribute. Every type may correspond to a number of attributes.
- *Convertibility*
  Each type definition is unique; there are no two type definitions carrying the same name or the same set of attributes.

An important property of this model, resulting from the relatability integrity rule, is that a given type can have different interpretations (e.g. attribute, aggregation, or generalization) with regard to related types. We will not explain the data manipulation and constraint definition languages here because there are few and intuitive constructs. Examples used in the sequel should become clear from the associated explanations.

3.2 Syntax specification

Frequently in this thesis we have to specify syntax. Although we could have used standard Extended Backus-Naur Form (EBNF) for most specification purposes, we preferred to use the same language for syntax specification that we implemented as support language for encapsulation tasks. As this language is more verbose than the commonly used EBNF notation the syntax specifications should be more easily understood.

The syntax specification language allows to define both lexical properties and syntax in a single specification file. While at this point this is merely a matter of convenience, it will become essential when using the language to capture input and output syntaxes in a single specification. Simple keywords can be directly placed into the grammar as symbols. Token classes have to be defined before their use by means of regular expressions.

3.2.1 Lexical properties

A language specification consists of format free text. There are comments, string constants, patterns, identifiers, keywords, and a few special characters. Both kinds of "C++" comments, bracketed in "/*" and "*/", and from "//" to end-of-line, are recognized. String constants are enclosed in double quotes and support the usual "C" escape characters "\n" for line-feed and "\t" for tabulator. Patterns are regular expressions enclosed in angular brackets ("<", ">"). All regular expressions of the scanner generator tool *lex* [Lesk 75] can be used. Identifiers are case sensitive and have the usual "C" appearance <[_A-Za-z][_A-Za-z0-9]*>.

The following keywords are defined:

-export, -ignore, -left, -nonassoc, -pattern, -prec, -right, -separator, -syntax, -type, list, opt, prec, range, repeat, rule, syntax, token

These are the token definitions for the specification language:

```
token LINE_COMMENT -ignore <"//" .* $>
token BRACKETED_COMMENT -ignore <"/*" .* "*/">

token ID -pattern <[_A-Za-z] [_A-Za-z0-9]*>
token PATTERN -pattern <"<" [^>]+ ">">
token LITERAL -pattern <\" ( [^"\\] | \\[nt\\"] )* \">
```

3.2.2 Constructs for Extended Backus-Naur Form

The specification language supports the following EBNF constructs:

```
rule word {
  | "opt" alternatives
  | "list" separator alternatives
  | "repeat" separator alternatives
}
rule separator {
    opt { "-separator" LITERAL }
}
```

Lists may define a separator literal like "," or ";". All EBNF constructs can be replaced with non-extended constructs.

- Optional parts are enclosed in curly braces and prefixed with the keyword *opt*. The replacement is:
  ```
  rule opt { /**/ | alternatives }
  ```
- Optional lists are enclosed in curly braces and prefixed with the keyword *list*. The replacement for lists without separator is:
  ```
  rule list { /**/ | list alternatives }
  ```
  Lists with separators are resolved as follows:
  ```
  rule sep_list { /**/ | tmp }
  rule tmp { alternatives | tmp separator:LITERAL alternatives }
  ```


- Lists with one or more elements are enclosed in curly braces and prefixed with the keyword *repeat*. The replacement for lists without separator is:
  ```
  rule repeat { alternatives | repeat alternatives }
  ```
  Lists with separators are resolved as follows:
  ```
  rule sep_repeat {
    alternatives | sep_repeat separator:LITERAL alternatives
  }
  ```

3.2.3 Structure of a language specification

A language specification consists of one or more named specification modules. Each module defines some comment conventions, lexical tokens, symbol precedence, and grammar rules:[*]

```
syntax meta_spec {
  rule modules {
        repeat { "syntax" ID "{" list { token | prec } repeat { rule } "}" }
  }
} // syntax meta_spec
```

3.2.4 Declarations

Declarations are used to define lexical language properties and symbol precedence. Token definitions associate a regular expression with a grammar symbol. The property *-ignore* may be used to declare a token as comment.

```
rule token {
    "token" ID opt { "-ignore" } "-pattern" PATTERN
}
```

Symbol precedence and associativity may be defined by a number of precedence statements, each with a list of literals or grammar symbols. Precedence and associativity are used in the generated parser to resolve grammar ambiguities. The earlier precedence statements list the literals and grammar symbols with low precedence. The later a precedence statement appears, the higher the precedence of its literals and grammar symbols.

[*] When presenting bits of syntax specification, we will frequently only use single modules or even only a few rules and will then omit the module header for simplicity.


```
rule prec {
    "prec" prec_property list { ID | LITERAL }
}
rule prec_property {
    "-left" | "-right" | "-nonassoc" | /**/
}
```

3.2.5 Grammar rules

The main part of a language specification consists of a list of grammar rules. A grammar rule specifies a left-hand side and a list of alternatives as right-hand sides. Each alternative in turn may consist of a list of words.

```
rule rule {
    "rule" ID alternatives
}
rule alternatives {
    "{" list -separator "|" { words } "}"
}
```

A word may either be

- a literal or a pattern,
- an identifier that denotes a token or a rule,
- an optional list of alternative words,
- a list (with either zero or one as the lower element count) of alternatives,
- or a subordinate set of alternatives.

Words may be tagged to give them a unique name for reference. A rule alternative may be assigned a precedence by naming a grammar symbol in the *-prec* property. The alternative receives the same precedence as was assigned to this symbol in a *prec* statement.

```
rule words {
    list { opt { ID ":" } word } opt { "-prec" ID }
}
```

```
rule word {
   PATTERN
 | ID | LITERAL
 | "opt" alternatives
 | "list" separator alternatives
 | "repeat" separator alternatives
 | alternatives
}
rule separator {
   opt { "-separator" LITERAL }
}
```

The syntax specification language as presented in this section will be extended by some specific constructs to enhance its usability as specification language for design file processors in Section 6.4 on page 111. There, we will also look into how efficient design file processors can be generated mostly automatically from such a specification.