T U M

INSTITUT FÜR INFORMATIK

Description Techniques for Data in the SysLab Method

R. Hettler

TUM-I9632
Oktober 1996

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-10-1996-I9632-350/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Description Techniques for Data in the SysLab Method

R. Hettler

Forschungsinstitut für
Angewandte Software-Technologie (FAST) e.V.,
Arabellastr. 17,
D-81925 München

Email: rhe@fast.de

October 8, 1996

Abstract

The SysLab method [Pae95], which is currently under development at the Technische Universität München, provides description techniques for the early phases of software engineering (requirements analysis, requirements definition and definition of the logical architecture). This report presents the techniques provided by the method for describing data. It considers the motivation for these techniques as well as their specific syntax and formal semantics.

Contents

1 Introduction

1.1 The SysLab Project

The SysLab project carried out at the TU München aims at a scientifically based approach for software- and system development. The emphasis in SysLab is on the early stages of software engineering: requirements analysis, requirements definition and logical design of systems, including prototyping.

In the course of the project a methodology (the SysLab method) is to be defined, which supports the development stages mentioned above. As usual, this methodology consists of a *process model*, which leads the developer through the different development stages, and a set of *description techniques*, which allow the developer to describe different *views* of the intended system, as required by the process model.

The description techniques provided by the SysLab method all possess a complete formal foundation, which is given in a very characteristic way. The formalization of all description techniques is based on a so-called *mathematical system model* [RKB95]. This system model formalizes the term *system*, as it is understood in SysLab. A system is seen as a set of components communicating messages with each other and the system environment via directed channels. Each component has itself the characteristics of a system, i.e. it can be subdivided into a set of subcomponents. The system model contains a notion of *discrete time*. Components have an internal *state*. The output of a component is determined by the communication history on its input channels and by its internal state. The system model is called *mathematical* because it is defined in a completely formal way, mainly based on the theory of *stream processing functions* [BDD+93].

As usual in software engineering methods, none of the provided description techniques is suited to describe the intended system as a whole. Rather, each of them concentrates on a certain view of the system. The description of the whole system consists therefore of a set of different documents, each of them describing a certain aspect of the system. As the semantics of each description technique is defined based on the mathematical system model, the contribution of the technique to the description of the whole system is made clear in a formal (and thus unambiguous) way. This approach allows formal reasoning about the following topics:

Expressiveness The question to be answered here is whether a given set of description techniques (e.g. all description techniques provided by the method) is sufficient to completely describe all aspects of the system as defined by the system model.

Consistency The views on the system which are provided by the description techniques are usually not disjoint. Often several description techniques contribute to the specification of a certain aspect of the system. An example is the description of the data of a system. When describing this static view (for example by drawing an entity-relationship diagram) one expresses certain consistency constraints for the data (such as cardinalities of relationship types). The specification of those consistency constraints has side effects on the dynamic behaviour of the system, however, because all the system functions must ensure not to violate the data consistency.

The approach taken in SysLab allows to formally identify such redundancies and provides thus a scientific basis for defining consistency and plausibility checks between different description techniques.

1.2 Description Techniques for Data

In the report at hand we deal with the data oriented part of the SysLab method. We present the description techniques provided by SysLab to specify the data of a system. For each description technique we will give

- an informal motivation,
- a definition of the syntax, and
- a formal semantics based on an algebraic specification language.

In a separate section the description techniqes will then be related to the mathematical system model.

2 Abstract Data Types

Many description techniques refer to primitive data elements, which are not further defined in the technique itself. Examples for such primitive data elements are attributes in entity-relationship modeling or messages exchanged between system components. In order to describe the domains of those primitive data elements, the SysLab method provides a description technique for *abstract data types*. An abstract data type provides data sorts and their characteristic operations through a well-defined interface. There are a lot of possible notations to specify abstract data types. The description technique Mini-Spectrum chosen in SysLab stems from the tradition of algebraic specification languages which started some twenty years ago [GTWW75, Gut75]. Mini-Spectrum is characterized by the following main features:

- It is a declarative language. The language for specifying the properties of sorts[1] and functions is mainly first order predicate logic which is expressive enough to give abstract and problem-oriented declarative specifications. Of course, specifications in this language are not necessarily executable in the sense of programs.

- It has a loose semantics, which makes it possible to work with underspecification. Underspecification means that it is not necessary to specify an abstract data type completely in one step. Instead it is possible to give a specification of an abstract data type which fixes only few important properties of the type and leaves the rest unspecified. The consequence is that for such a specification several different implementations (models) are possible. The set of possible implementations can later be restricted by making the specification more detailed, for instance by adding new properties of the type.

MINI-SPECTRUM is closely related to the algebraic specification language SPECTRUM [BFG+93a, BFG+93b, GN94]. In fact, it can be seen as a sublanguage of SPECTRUM in the sense that all the concepts present in MINI-SPECTRUM can also be found in SPECTRUM. MINI-SPECTRUM is not as expressive as SPECTRUM, but has, on the other hand, the advantage of being much easier to learn and more comfortable to use. We consider MINI-SPECTRUM to be powerful enough for almost all applications which can be modeled in the SYSLAB method. However, if an experienced user of the method should find it necessary to specify an abstract datatype which is too complex to be adequately described using MINI-SPECTRUM, he can use SPECTRUM for this task. This is due to the fact that SPECTRUM is used as the formal foundation for specifying abstract data types and MINI-SPECTRUM is introduced simply as sublanguage thereof.

In the next sections we will first present the syntax[2][3] and (informally) the meaning of the language constructs of MINI-SPECTRUM together with some examples, and then define the semantics of the language formally by giving a translation into SPECTRUM, which itself has of course a formal semantics.

---

[1]The term 'sort' is in the field of algebraic specification techniques used for what is usually called 'type' in programming languages, which is a name for a set of elements. This is done in order not to confuse sorts with abstract data types, which often are also simply called types.

[2]The syntax of the language is given in EBNF (Extended Backus-Naur Form). We consider this notation to be self-explanatory. A description can be found in [BFG+93b], where the syntax of SPECTRUM is defined.

[3]We will not present the lexical syntax (legal identifiers, comments, ...) of the language here. This is defined to be the same as the lexical syntax of SPECTRUM (see [GN94]). Presenting the lexical syntax here as well would not contribute to understanding the concepts of MINI-SPECTRUM.

2.1 Informal Description of Mini-Spectrum

$$\langle\text{specification}\rangle \quad ::= \quad \langle\text{spec-id}\rangle = [\ [\langle\text{enriches}\rangle]\ \langle\text{signature}\rangle\ \langle\text{axioms}\rangle\ ]$$

In order to be able to distinguish them from Spectrum specifications, specifications in Mini-Spectrum are enclosed in square brackets (instead of braces). Each Mini-Spectrum specification has a unique name (⟨spec-id⟩) and consists of three parts:

- an optional imports section (⟨enriches⟩), in which the signatures of primitive specifications can be imported,
- a signature section (⟨signature⟩), in which new sort and function symbols can be introduced, and
- a properties section (⟨axioms⟩), in which the properties of the introduced sorts and functions are logically specified.

2.1.1 Using Primitive Specifications

$$\begin{array}{lcl} \langle\text{enriches}\rangle & ::= & \textbf{enriches}\ \langle\text{specimp}\rangle\ \{+\ \langle\text{specimp}\rangle\}^{*} \\ \langle\text{specimp}\rangle & ::= & \langle\text{spec-id}\rangle\ [[\ \{\langle\text{ident}\rangle\ //\ \mathbf{,}\}^{+}\ ]] \\ \langle\text{ident}\rangle & ::= & \langle\text{id}\rangle\ \textbf{to}\ \langle\text{id}\rangle \\ & \mid & \langle\text{inf-id}\rangle\ \textbf{to}\ \langle\text{inf-id}\rangle \\ & \mid & \langle\text{sid}\rangle\ \textbf{to}\ \langle\text{sid}\rangle \end{array}$$

Mini-Spectrum specifications can be hierarchically based on other Mini-Spectrum specifications, which are then called *primitive specifications*. All the primitive specifications of a Mini-Spectrum specification are listed in its imports section. This makes the symbols (sorts and functions) defined in the primitive specifications accessible in the current specification. Those primitive symbols are, however, not added to the signature and are thus not exported by the specification.

2.1.2 Defining Signatures

$$
\begin{array}{lcl}
\langle\text{signature}\rangle & ::= & \{\langle\text{sort}\rangle \mid \langle\text{function}\rangle \mid \langle\text{constr}\rangle\}^{*} \\
\\
\langle\text{sort}\rangle & ::= & \textbf{sort}\ \langle\text{sortcon}\rangle\ \{\langle\text{sortvar}\rangle\}^{*} \\
 & \mid & \textbf{data}\ \langle\text{sortcon}\rangle\ \{\langle\text{sortvar}\rangle\}^{*} = \{\langle\text{product}\rangle\ //\ |\}^{+} \\
\langle\text{product}\rangle & ::= & \langle\text{id}\rangle \\
 & \mid & \langle\text{opn}\rangle\ \textbf{(}\ \{[\langle\text{id}\rangle\ \textbf{:}]\ \langle\text{sortexp}\rangle\ //\ \textbf{,}\}^{+}\textbf{)} \\
\\
\langle\text{function}\rangle & ::= & \langle\text{opns}\rangle\ \textbf{:}\ \langle\text{sortexp}\rangle \\
\langle\text{sortexp}\rangle & ::= & \langle\text{sortexp1}\rangle \\
 & \mid & \langle\text{sortexp1}\rangle \rightarrow \langle\text{sortexp}\rangle \\
\langle\text{sortexp1}\rangle & ::= & \langle\text{sortexp2}\rangle \\
 & \mid & \langle\text{sortexp2}\rangle\ \{\times\ \langle\text{sortexp2}\rangle\}^{+} \\
\langle\text{sortexp2}\rangle & ::= & \langle\text{asort}\rangle \\
 & \mid & \langle\text{sortcon}\rangle\ \{\langle\text{asort}\rangle\}^{*} \\
\langle\text{asort}\rangle & ::= & \langle\text{sortvar}\rangle \\
 & \mid & \textbf{(}\ \langle\text{sortexp}\rangle\ \textbf{)} \\
\\
\langle\text{constr}\rangle & ::= & \langle\text{sortcon}\rangle\ \{\langle\text{sortvar}\rangle\}^{*}\ \textbf{generated by}\ \langle\text{opns}\rangle \\
\\
\langle\text{opn}\rangle & ::= & \langle\text{id}\rangle \\
 & \mid & \textbf{.}\langle\text{id}\rangle\textbf{.} \\
\langle\text{opns}\rangle & ::= & \{\langle\text{opn}\rangle\ //\ \textbf{,}\}^{+}
\end{array}
$$

The signature section of a Mini-Spectrum specification allows to introduce new sorts (or rather sort constructors) and functions. Function symbols are introduced together with their functionality. Furthermore, some of the introduced functions can be distinguished to be *constructor functions* for the newly introduced sorts.

Sorts Mini-Spectrum provides a polymorphic sort system very similar to that known from functional programming languages. Sorts are interpreted as sets of elements. Mini-Spectrum allows to define *sort constructors* in its signature part. Sort constructors are functions on the sort level. Hence they yield sorts when applied to sorts. Sort constructors can have arbitrary arity. A sort constructor of arity 0 corresponds to a sort. With this concept of sort constructors it is for example possible to introduce a sort (0-ary sort constructor) `Nat` and a unary sort constructor `Set` α (The symbol α is a formal parameter and indicates that the sort constructor `Set` is of arity 1). We can now build new sorts by applying the sort constructor `Set`: `Set Nat, Set(Set Nat)`, ...[4]

[4]Note that in Mini-Spectrum sort variables must not be instantiated with functional sorts. The sort expression `Set(Nat` $\longrightarrow$ `Nat)` is thus forbidden. This restriction in polymorphism is

Sort constructors can be introduced either with the `sort` construct or with the `data` construct of MINI-SPECTRUM. The `sort` construct simply introduces the sort constructor symbol without specifying any properties of this constructor at all. This means that the properties of the sort constructor have to be specified via the properties of functions operating on sorts constructed with this constructor.

The `data` construct, on the other hand, introduces a new sort constructor and already completely fixes its properties. Sort constructors introduced with the `data` construct construct so-called *free data types* as known from functional languages. The `data` construct therefore corresponds for example closely to the `datatype` construct provided by the functional programming language ML [HMM86]. Using the `data` construct we can for example specify a sort constructor `List` which, when applied to an arbitrary sort α, yields the sort of all finite sequences of elements of sort α:

```
data List α = empty | cons(first:α, rest:List α);
```

This line introduces

- The unary sort constructor `List`.
- Constructor functions `empty` and `cons` for sorts constructed with this sort constructor. For those two functions the following properties hold:
  1. All elements of the sort can be represented by a constructor term (i.e. a term which contains only applications of the constructor functions).
  2. Different constructor terms denote different elements (lists).
- Selector functions `first` and `rest` which allow to access the two components of lists constructed with `cons`.

Functions The second kind of symbols which can be introduced in the signature part are function identifiers. Functions defined in MINI-SPECTRUM are strict[5] by definition. They are not necessarily total[6], although the axiomatization in the properties part can imply totality, of course.

For each function symbol its functionality is defined, fixing the sorts of its arguments and the sort of its result. For example, the function

due to the specific selection of the part of SPECTRUM which constitutes MINI-SPECTRUM. The alternative to this decision would be to introduce some kind of class system over sorts which would make the whole language considerably more complex.

[5]A strict function yields an undefined value whenever one of its arguments is undefined. The notion of undefinedness is introduced in MINI-SPECTRUM to model diverging calculations, i.e. nontermination. In MINI-SPECTRUM each sort contains therefore one distinguished element $\perp$ which represents the undefined value.

[6]A total function always yields a defined result when its arguments are defined.

`add :  Nat` $\times$ `Nat` $\longrightarrow$ `Nat`

takes two elements of sort `Nat` as arguments and yields a result which is also of sort `Nat`. Note that there are also functions of arity 0, which represent constants. If a function has exactly two arguments, it may be defined to be an infix function by enclosing it between dots which symbolize the argument positions:

`.+.  :  Nat` $\times$ `Nat` $\longrightarrow$ `Nat`

In MINI-SPECTRUM, functions may be higher order. This means that they may have functions as arguments or result. Thus, a functionality like the following is allowed:

`f :  (Nat` $\longrightarrow$ `Nat)` $\longrightarrow$ `(Nat` $\longrightarrow$ `Nat)`

Functions may furthermore be polymorphic, making use of the concept of sort constructors explained above. For example, the function

`length :  List` α $\longrightarrow$ `Nat`

can be applied to any sort constructed with the sort constructor `List`. In MINI-SPECTRUM, greek letters are used to denote *sort variables*. The function `length` has a functionality which contains a sort variable α. It is therefore generic in the sense that it can applied to arguments of any sort which can be obtained by instantiating the sort variable with a specific sort.

Constructor Functions MINI-SPECTRUM allows to distinguish a set of functions to be constructors of a given sort. This means that every element of the sort can be denoted by a closed term which contains only functions from the constructor set. We can, for example, specify the constructors of the natural numbers by

```
Nat generated by 0, succ;
```

This means that every natural number can be constructed by applying only the functions `0` and `succ`[7].

[7]If a sort is introduced with the `data` construct, the constructor functions are already determined by this construct. In this case, there is no need to give an additional `generated by` statement.

2.1.3 Specifying Properties

$$
\begin{array}{lcll}
\langle\text{axioms}\rangle & ::= & \textbf{axioms}\ [\langle\text{varlist}\rangle]\ \{\langle\text{formula}\rangle\ \textbf{;}\}^{*}\ \textbf{endaxioms} & \\
 & & & \\
\langle\text{varlist}\rangle & ::= & \forall\ \langle\text{opdecls}\rangle\ \textbf{in} & \\
\langle\text{opdecls}\rangle & ::= & \{\{\langle\text{id}\rangle\ //\ \textbf{,}\}^{+}\textbf{:}\ \langle\text{sortexp}\rangle\ //\ \textbf{,}\}^{+} & \\
 & | & \{\langle\text{id}\rangle\ //\ \textbf{,}\}^{+} & \\
 & & & \\
\langle\text{formula}\rangle & ::= & \{\forall \mid \exists\}\ \langle\text{opdecls}\rangle\ \textbf{.}\ \langle\text{form1}\rangle & \\
\langle\text{form1}\rangle & ::= & \langle\text{aform}\rangle & \\
 & | & \neg\ \langle\text{aform}\rangle & \\
 & | & \langle\text{aform}\rangle \Rightarrow \langle\text{form1}\rangle & \\
 & | & \langle\text{aform}\rangle \Leftrightarrow \langle\text{form1}\rangle & \\
 & | & \langle\text{aform}\rangle \wedge \langle\text{form1}\rangle & \\
 & | & \langle\text{aform}\rangle \vee \langle\text{form1}\rangle & \\
 & & & \\
\langle\text{aform}\rangle & ::= & \langle\text{exp}\rangle = \langle\text{exp}\rangle & \\
 & | & \textbf{(}\ \langle\text{formula}\rangle\ \textbf{)} & \\
 & & & \\
\langle\text{exp}\rangle & ::= & \langle\text{exp1}\rangle & \\
 & | & \langle\text{exp1}\rangle == \langle\text{exp1}\rangle & \\
\langle\text{exp1}\rangle & ::= & \langle\text{exp2}\rangle & \\
 & | & \langle\text{exp2}\rangle\ \langle\text{id}\rangle\ \langle\text{exp2}\rangle & \textit{(Infix Function Application)} \\
\langle\text{exp2}\rangle & ::= & \langle\text{aexp}\rangle & \\
 & | & \langle\text{exp2}\rangle\ \langle\text{aexp}\rangle & \textit{(Prefix Function Application)} \\
\langle\text{aexp}\rangle & ::= & \langle\text{opn}\rangle & \\
 & | & \perp & \\
 & | & \textbf{if}\ \langle\text{exp}\rangle\ \textbf{then}\ \langle\text{exp}\rangle\ \textbf{else}\ \langle\text{exp}\rangle\ \textbf{endif} & \\
 & | & \textbf{(}\ \langle\text{exp}\rangle\ \textbf{)} & \textit{(Grouping)} \\
 & | & \textbf{(}\ \langle\text{exp}\rangle\ \{\textbf{,}\ \langle\text{exp}\rangle\}^{+}\ \textbf{)} & \textit{(Tuples)}
\end{array}
$$

The properties part of a Mini-Spectrum specification serves for fixing the properties of the symbols introduced in the signature part. The language provided for this purpose is mainly typed first-order predicate logic. The properties are given as a set of logical formulae between the keywords **axioms** and **endaxioms**, which have to be true in all models (and also for all implementations) of the specification. The optional variable list ($\langle$varlist$\rangle$) gives a set of all-quantified variables, the scope of which is the whole set of formulae. This is simply a shorthand notation. We could as well repeat this all-quantification in front of every single formula.

Formulae Logical formulae are built in MINI-SPECTRUM from predicates over expressions using the quantifiers $\forall$ and $\exists$ as well as the logical combinators $\neg$, $\vee$, $\wedge$ and $\Rightarrow$. The only predicate provided by MINI-SPECTRUM is the so-called *strong equality* $=$, which is nonstrict, because it yields a defined value (true or false) even for undefined arguments[8].

Expressions Atomic expressions in MINI-SPECTRUM are function identifiers and the special symbol $\perp$ which denotes the undefined element. More complex expressions can be built from those using:

Function Application: Applying an expression of a function sort to argument expressions of sorts matching the parameter sorts yields an expression of the result sort of the function expression..

Weak Equality: For every sort except those involving the function sort constructor $\longrightarrow$ there is a Boolean function called weak equality and denoted by **.==.**, which is a strict and decidable equality predicate.

Case Distinction: For expressing case distinction MINI-SPECTRUM provides an **if**...**then**...**else**...**endif** construct.

Tuple Constructors: Tuple expressions can be built by grouping expressions using parentheses.

2.1.4 Built-In Specification

A sort representing Boolean values and functions operating on those values is predefined in MINI-SPECTRUM. They can be used in any MINI-SPECTRUM specification without the need for explicitly giving an **enriches** command. The definition of this abstract data type of Boolean values can itself be expressed in MINI-SPECTRUM notation and is given below.

```
Bool = [

data Bool = true | false;

not :  Bool ⟶ Bool;
.and., .or., .impl.  :  Bool × Bool ⟶ Bool;
```

[8] Readers familiar with SPECTRUM may notice that MINI-SPECTRUM clearly distinguishes the level of formulae from the level of terms. In SPECTRUM, on the contrary, this distinction is not made. This mixing of levels in SPECTRUM leads to a somewhat complex and sometimes clumsy looking three-valued logic. MINI-SPECTRUM therefore rather adopts the classical distinction between terms and formulae.

```
axioms ∀ x,y : Bool in
not(true) = false;
not(false) = true;

x and y = y and x;
true and x = x;
false and x = false;

x or y = y or x;
true or x = true;
false or x = x;

true impl x = x;
false impl x = true;
endaxioms
]
```

2.2 Example Specifications

In the following we give two example Mini-Spectrum specifications in order to illustrate the language definition given in the previous sections. We will give short explanations together with the specifications, although we believe them to be well understandable.

2.2.1 Natural Numbers

```
Nat = [

data Nat = 0 | succ(pred:Nat);

.+., .-., .*., .div., .mod. : Nat × Nat ⟶ Nat;
.≤., .<. : Nat × Nat ⟶ Bool;

axioms ∀ x,y : Nat in
0 ≤ x = true;
succ(x) ≤ succ(y) = x ≤ y;
succ(x) ≤ 0 = false;

x < y = (x ≤ y) and not(x == y);

x + 0 = x;
```

```
x + succ(y) = succ(x + y);

x - 0 = x;
succ(x) - succ(y) = x - y;
0 - succ(x) = ⊥;

x * 0 = 0;
x * succ(y) = (x * y) + x;

x div 0 = ⊥;
x mod 0 = ⊥;
(x mod succ(y)) < succ(y);
x = ((x/succ(y)) * succ(y)) + (x mod succ(y));
endaxioms
]
```

`Nat` specifies the well-known concept of natural numbers. It first introduces the sort `Nat` using the `data` construct. This construct not only introduces the sort, but also its constructor functions `0` and `succ` and the selector function `pred` as the inverted function of `succ`.

The properties part then gives all the well-known properties of the functions working on natural numbers.

2.2.2 Polymorphic Finite Sets

```
Set = [

sort Set α;

∅ : Set α;
add, del : α × Set α ⟶ Set α;
.∈. : α × Set α ⟶ Bool;

Set α generated by ∅, add;

axioms ∀ x,y : α, s : Set α in
add(x,add(x,s)) = add(x,s);
add(x,add(y,s)) = add(y,add(x,s));

x ∈ ∅ = false;
x ∈ add(y,s) = (x == y) or (x ∈ s);
```

```
x ∈ del(x,s) = false;
not(x==y) ⇒ x ∈ del(y,s) = x ∈ s;
endaxioms
]
```

The most remarkable aspect about the specification `Set` is that it is polymorphic. Hence, using the unary sort constructor `Set` α defined in this specification, we can build set sorts over arbitrary sorts (except functional sorts). The functions $\emptyset$ and `add` are defined as set constructors. This indicates that all sets can be described using terms over those two functions. The properties part describes the properties characteristic for finite sets in a polymorphic way. This is possible because the characteristics of finite sets are independent of the element sort. The first two formulae give the typical set properties (elements cannot occur more than once in a set and the order of insertion of elements into a set is irrelevant). The remaining formulae describe the characteristics of the non-constructor functions `.∈.` and `del`.

A further aspect worth noting is that the sort constructor `Set` α could not have been specified using the `data` construct of Mini-Spectrum. Sort constructors specified using `data` are so-called *free data types*, which means that different constructor terms always denote different semantic elements. This is not the case with sets, as can be seen from the the first two axioms in the properties part of the specification.

2.3 Semantics of Mini-Spectrum

As explained above, Mini-Spectrum constitutes a sublanguage of the algebraic specifications language Spectrum. It is a proper subset of Spectrum in the following sense:

- The language for specification in the large is restricted compared to Spectrum. Mini-Spectrum provides the concept of hierarchically basing a specification on primitive specifications (including renaming of signature elements), but it has no notion for hiding signature elements and for parametrization of specifications.
- Mini-Spectrum has, in contrast to Spectrum, no system of sort classes.
- Spectrum's language for specifying in the small provides a lot of concepts which are, for the sake of simplicity, not present in Mini-Spectrum: sort synonyms, **let**- and **letrec**-expressions, lambda abstraction, a built-in fix point operator, built-in mixfix syntax for lists, ...

All those simplifications make Mini-Spectrum a very lean and easy-to-use algebraic specification language without affecting expressivity and flexibility too much. Besides that Mini-Spectrum makes some semantic assumptions about sorts and functions which further simplify its use and which are not there in Spectrum where they have to be made explicit in the specification.

- All functions introduced in a Mini-Spectrum specification are assumed to be strict. In Spectrum, strictness of a function has to be specified explicitly.
- All sorts except functional sorts are assumed to denote flat domains. In particular, user-defined sort constructors can only define flat sorts. In Spectrum the requirement of denoting a flat domain has to be expressed explicitly, using the concept of sort classes (the sort class EQ represents the set of all flat sorts in Spectrum).

From the above it is clear that everything which can be expressed in Mini-Spectrum can also be expressed in Spectrum. The easiest way to define the semantics of Mini-Spectrum is therefore to give a translation from Mini-Spectrum specifications to Spectrum specification. Thus, the formal semantics of Spectrum [GR94] is used to assign meaning to Mini-Spectrum specifications.

2.3.1 Translation Mini-Spectrum $\longrightarrow$ Spectrum

The syntax of Mini-Spectrum is already very close to that of Spectrum. Therefore, in order to map a Mini-Spectrum specification S into the corresponding Spectrum specification, only the following modifications have to be applied to S:

1. The square brackets enclosing the specification have to be replaced by braces:

   `SP = [...]` $\leadsto$ `SP =` $\{\ldots\}$

2. Mini-Spectrum uses a shorter syntax for enrichment with renaming of signature elements than Spectrum:

   `enriches ...  [...]  ...` $\leadsto$ `enriches rename ...  by [...]  ...`

3. Spectrum does not distinguish between the level of formulae and the level of Boolean terms. The Boolean functions `not`, `.and.`, `.or.` and `.impl.` are therefore not predefined in Spectrum as they are in Mini-Spectrum. The predefined Boolean operators in Spectrum, however, can take the

role of those symbols, because they are applicable on the level of formulae as well as on the level of terms:

`not`	$\leadsto$	$\neg$
`.and.`	$\leadsto$	$.\wedge.$
`.or.`	$\leadsto$	$.\vee.$
`.impl.`	$\leadsto$	$.\Rightarrow.$

4. In Mini-Spectrum all functions are assumed to be strict. In Spectrum this property has to be demanded explicitly. This can be done by inserting the keyword **strict;** anywhere in the specification (but after the **enriches** section):

   ```
   SP = [ enriches ...; ...]
             ⤳
   SP = { enriches ...; strict; ...}
   ```

   This keyword demands strictness of all functions which are newly introduced in a Spectrum specification except constructor functions defined within a **data** construct. For this kind of functions strictness has to be demanded within the **data** construct by placing exclamation marks in front of all argument positions in which the constructor function is to be strict:

   ```
   SP = [...
   data List α = nil | cons(first :  α, rest :  List α);
   ...]
             ⤳
   SP = {...
   data List α = nil | cons(!first :  α, !rest :  List α);
   ...}
   ```

5. In Mini-Spectrum all sorts (except functional sorts) represent flat domains. While this is implicit in Mini-Spectrum, it has to be explicitly demanded in Spectrum. For this purpose, Spectrum provides the sort class `EQ`. When translating a Mini-Spectrum specification to Spectrum, this means first of all that we have to specify the correct class membership along with each newly introduced sort constructor (no matter if it is introduced via **sort** or **data**):

   ```
   sort Nat;  ⤳  sort Nat; Nat::EQ;
   data List α = ... ⤳  data List α = ...; List::(EQ)EQ;
   ```

   Furthermore, all occurences of sort variables have to be restricted to the class `EQ` using so-called *contexts*. This applies to the signatures of polymorphic functions as well as to polymorphic sorts of variables in formulae:

- $\texttt{f} : \alpha \longrightarrow \beta \leadsto \texttt{f} : \alpha,\beta\texttt{::EQ} \Rightarrow \alpha \longrightarrow \beta$
- Let $\alpha_1, \ldots, \alpha_n$ be the set of sort variables occurring in the properties section of a MINI-SPECTRUM specification. Then

  $\texttt{axioms}\ \forall\ \ldots\ \texttt{in}\ \ldots\ \texttt{endaxioms}$

  $\leadsto$

  $\texttt{axioms}\ \alpha_1,\ldots,\alpha_n\texttt{::EQ} \Rightarrow \forall\ \ldots\ \texttt{in}\ \ldots\ \texttt{endaxioms}$

2.3.2 Example

The MINI-SPECTRUM specification of polymorphic finite sets of Section 2.2.2 translates according to the rules given above to the following SPECTRUM specification:

```
Set = {

sort Set α;
Set::(EQ)EQ;

∅ :  α::EQ ⇒ Set α;
add, del :  α::EQ ⇒ α × Set α ⟶ Set α;
.∈.  :  α::EQ ⇒ α × Set α ⟶ Bool;

Set α generated by ∅, add;

axioms α::EQ ⇒ ∀ x,y :  α, s :  Set α in
add(x,add(x,s)) = add(x,s);
add(x,add(y,s)) = add(y,add(x,s));

x ∈ ∅ = false;
x ∈ add(y,s) = (x == y) or (x ∈ s);

x ∈ del(x,s) = false;
not(x==y) ⇒ x ∈ del(y,s) = x ∈ s;
endaxioms;
}
```

This specification is used to give meaning to the MINI-SPECTRUM specification of Section 2.2.2. This means that the model class of the MINI-SPECTRUM specification is defined to be the same as the model class of the SPECTRUM specification given here.

3 Extended Entity-Relationship Modeling

Information systems usually administer complexely structured mass data. The task of describing these data structures, normally called *(conceptual) data modeling*, plays therefore an important role in information system modeling. In principle, this task could be performed using a description technique for abstract data types, for example the language Mini-Spectrum presented in Section 2. It is however clear that such a technique is not very well suited to gain intellectual control over very complex data structures which usually occur in data modeling. Software Engineering methods provide therefore specific, mostly graphical, description techniques for this purpose, the most widespread and popular of which is the *entity-relationship model (ERM)* [Che76].

This description technique structures the data which are to be modeled into meaningful[9] units, called *entities* and classifies these entities using *entity types.* An entity type is described using *attributes.* Attributes are identifiers for data values which are used to characterize entities of the respective type. Attributes can be mandatory or optional. A mandatory attribute must always contain a value, while an optional attribute is allowed not to contain any value out of the given domain. Entities can be related to other entities via *relationships.* Relationships are classified using *relationship types.* A relationship type is characterized by the entity types of the entities which participate in relationships of that type. The participation of an entity type in a relationship type is usually called *role.* Entity types may participate in relationship types in more than one role. The data structure consisting of entity types, attributes and relationship types is usually denoted graphically in an *entity-relationship diagram (ERD).*

Since 1976 many different variants of the original entity-relationhip model have been developed which differ in their graphic representation as well as in their expressivity. In the following we present a variant of this technique tailored for the SysLab project, which includes one extension of the technique which is not present in any of the other variants.

3.1 Entity-Relationship Diagram Name

In our approach an every entity-relationship diagram has a unique name, which is given on top of the diagram as a title. This name allows to refer to an ERD from other description techniques of the method.

[9]Meaningful with respect to the application domain. Some piece of information may be considered a meaningful unit in one application, while in another it would never occur alone but always as part of some other, larger block of information, which would be a meaningful unit of information for this second application.

3.2 Entity Types and Attributes

Entity types are characterized by their attributes. An attribute has a type (*attribute type*), which denotes the domain (the set of allowed values) of the attribute. Entity types are represented as labelled rectangular boxes in the entity-relationship diagram. An entity type is not further described in the ERD. The description of an entity type is given in a so-called *entity type description table*. Examples of such tables are contained in the example data schema given in Section 3.4.4. For each entity type in the ERD there is exactly one entity type description table, which contains for every attribute of the type the following information:

- The name of the attribute.
- The type of the attribute, which fixes the domain of the attribute's values. For the sake of completeness we require every attribute type to be specified as a sort in an abstract data type description[10] (see Section 2).
- Information about whether the attribute is mandatory or optional.
- Information about whether the attribute is part of the entity type's key (cf. Section 3.4.1).

3.3 Relationship Types

Relationship types in our approach may have an arbitrary arity n ($n > 1$). A relationship type is depicted in the ERD as a labelled rhombus, which is connected to the participating entity types via solid lines (cf. Section 3.4.4). Each line represents a role and can be labelled with a role identifier. If an entity type participates in a relationship type in more than one role, the roles have to be labelled.

3.4 Static Integrity Constraints

The entity-relationship model allows to describe a data structure using entity types, attributes and relationship types. Besides the purely structural aspects, however, the ERM provides a notation for certain kinds of constraints which the modeled data have to obey. This kind of constraints is called *static integrity constraints*, because they are used to specify data integrity and because they have a static nature, which means that they do not allow to express any restrictions about data evolution over time. The classical ERM provides only two

[10] If an attribute type T is not yet specified by the developer in an abstract data type we assume it to be defined by the simple MINI-SPECTRUM specification `T = [ sort T; ]`.

specific kinds of static integrity constraints: keys of entity types and cardinalities of relationship types. The entity-relationship model provided by SysLab goes significantly beyond this level by providing the full expressive power of first-order predicate logic for specifying static data integrity.

3.4.1 Keys

A key is a subset of the attributes of an entity type, which suffices to uniquely characterize the entities of that type. The constraint connected with a key is therefore that there must not exist two different entities with identical key attribute values. The attributes forming the key of an entity type are in our notation marked in the respective entity type description table (cf. Section 3.4.4).

3.4.2 Cardinalities

Cardinalities are constraints which restrict the participation of entities in relationship types. There are several notations for cardinalities. A very common notation is to classify relationship types to be 1:1, 1:n or m:n. This classification scheme only works for binary relationship types. It is not appropriate for the description technique presented here. We therefore adopt the so-called *(min,max) notation* (cf. for example [SS83]) to express cardinalities. This notation works for relationship types of arbitrary arity. In this notation, every role, i.e. every line connecting an entity type with a relationship type in the ERD, is annotated with a tuple (*min*,*max*) which specifies the minimal and maximal number of participations of entities of the given entity type relationships of the given relationship type in that specific role. The second component of this tuple may also be an asterisk $*$ which indicates that there is no restriction concerning the maximum number of participations. For examples again see the ERD given in Section 3.4.4.

3.4.3 General Constraints

In conceptual data modeling, one often finds static integrity constraints for the data to be modeled which cannot be expressed using key and cardinality constraints. In order to record those more complex constraints, too, the SysLab variant of the entity-relationship model offers the possibility to annotate the ERD with logic formulae which express those constraints. It should be noted that the above mentioned key and cardinality constraints can of course be expressed as formulae, too. The notations for keys and cardinalities can thus be seen as graphic abbreviations for special kinds of formuale.

As a language for those formulae we chose Mini-Spectrum. Hence we can use the same kind of formulae to annotate an ERD as can be found in the properties section of a Mini-Spectrum specification. The formulae are expressed over a signature which is generated from the ERD in the following way:

- Entity types represent sorts (*entity sorts*).
- Attributes are modeled as selector functions mapping entities to attribute values.
- Relationship types are seen as mathematical relations between entity types and modeled as Boolean functions which represent the characteristic predicates of those relations.

We do not go into further detail about this generated signature here. This topic will occur again in Section 3.5. An example for a static integrity constraint expressed as logic formula is contained in the entity-relationship schema given in Section 3.4.4, an example for a signature generated from an ERD can be found in Section 3.5.3.

3.4.4 Example

The entity-relationship schema of Figure 1 shows the data structure to be administered by a very simplistic bibliographic information system (BIS). This system allows to store and retrieve information about publications, their authors and publishers. Furthermore the system has to keep record of the literature references contained in the publications, such that a user of the system can find out which publications are referenced by a certain publication. As a last feature the system incorporates a glossary of keywords used to characterize the publications, which means that each publication can be linked with one or more keywords and that for each keyword a definition text is kept in the system.

We believe the entity-relationship diagram and the entity type description tables given in Figure 1 to be self-explanatory. We will therefore only explain the formula given under the headline 'Static Integrity' in this figure. The schema of our bibliographic information system contains a special kind of redundancy concerning publishers. Information about publishers is modeled in our schema as entity type which is connected to its publications via the relationship type publishes. If a publication has an ISB-number, however, publisher information is also contained in the ISBN (each ISBN contains a 3-digit code which identifies the publisher of the publication). For publications with an ISBN publisher information is therefore contained twice in the schema. Of course, it is a necessary condition that this redundant information is always consistent in the sense that the publisher code in the ISBN describes the same publisher as the publisher entity related to the publication entity via publishes. It is exactly this constraint which is expressed by the Mini-Spectrum formula in Figure 1. Informally, this

BIS

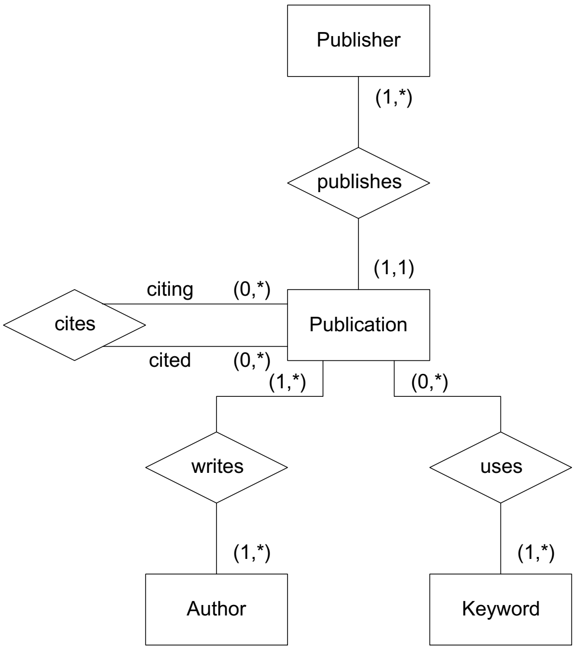
Primitive Data Types:

```
Isbn = {
sort Isbn;
}
```

```
Adr = {
data Adr = mkadr(!Country : String,
                 !City    : String,
                 !Street  : String,
                 !ZIP     : Nat);
}
```

```
Date = {
data Date = mkdate(!Day   : Nat,
                   !Month : Nat,
                   !Year  : Nat);
}
```

Structure:



Entity Type Descriptions:

Entity Type Publisher			
Attribute	**Type**	**Opt**	**Part of Key**
P_Name	String		•
P_Address	Adr		

Entity Type Publication			
Attribute	**Type**	**Opt**	**Part of Key**
Title	String		•
Year	Nat		•
ISBN	Isbn	•	

Entity Type Author			
Attribute	**Type**	**Opt**	**Part of Key**
Name	String		•
Dateofbirth	Date		•
Address	Adr		

Entity Type Keyword			
Attribute	**Type**	**Opt**	**Part of Key**
Notion	String		•
Description	String		

Static Integrity:

```
∀P:Publisher,p:Publication.
   publishes(P,p) ⇒ ISBN(p)=NULL ∨ consistent(P_Name(P), ISBN(p))
```

Figure 1: Example Entity-Relationship Schema

formula says: For every pair of a publisher entity `P` and a publication entity `p` which are related via `publishes` the ISBN attribute is either `NULL`[11] or it is consistent with the publisher entity `P`. The consistency check itself takes place in an auxiliary function `consistent`, which is not specified here.

3.5 Semantics of the Extended Entity-Relationship Model

For the different variants of the ERM there are already many semantics definitions, informal ones (given in natural language) as well as mathematical ones, for example by translating entity-relationship schemas to the relational calculus. As most of those ER variants do not possess the extension described above, their semantics definitions also do not discuss this extension[12].

In the following we sketch the definition of a formal semantics for the extended entity-relationship approach presented in this paper. A thorough discussion of this way of giving a formal semantics to the ERM can be found in [Het95].

In this approach an entity-relationship schema is viewed as the definition of an abstract data type. This is done by giving translation rules which allow to assign a Mini-Spectrum specification[13] to each ER schema. The abstract data type described by this specification is then defined to be the semantics of the ER schema, too.

3.5.1 Translating Extended Entity-Relationship Schemata into Mini-Spectrum

In the following we present the definition of a translation schema for translating ERDs into Mini-Spectrum specifications. A discussion of this topic in full detail is out of the scope of this paper. Instead, we present the main aspects of this schema and illustrate them by an example. For further information we refer to [Het95].

Signature As we have already stated in Section 3.4.3, an entity-relationship diagram can be seen as graphical definition of a signature, where entity types modeled as sorts, attributes as selector functions on entity sorts and relationship types as mathematical relations. The signature reflecting the structure of

[11]The special value NULL is used to model the fact that an optional attribute carries no defined value.

[12]There are, however, some approaches to entity-relationship modeling which contain extensions similar to the one given here and which possess a formal semantics definition (cf. [JRP91a, JRP91b, Gog94, Hoh93]).

[13][Het95] uses the language Spectrum for this purpose, but the sublanguage of Spectrum comprised by Mini-Spectrum suffices.

the schema of the bibliographic information system presented in Section 3.4.4 is shown in the specification given in Section 3.5.3 and certainly suffices to illustrate the idea behind the generation of signatures from entity-relationship schemata.

Properties The signature of the generated specification reflects the structure of the data modeled by the corresponding schema. The static integrity constraints are consequently represented by the axioms in the properties part of the specification, thus giving the restrictions which the signature elements have to obey. Concerning the semantics of static integrity constraints, we have to distinguish three cases:

- **Key constraints** are in the presented approach given in the entity type description tables. Their meaning can, of course, be also expressed as MINI-SPECTRUM formulae over the above signature.

  Suppose an entity type `E` with attributes $\mathtt{a}_1$, ..., $\mathtt{a}_n$, where $\mathtt{a}_1$, ..., $\mathtt{a}_k$ ($k \leq n$) are the key attributes. The key constraint connected with this key is then expressed by the MINI-SPECTRUM formula

  $$\forall \mathtt{e}_1,\mathtt{e}_2\mathtt{:E.}\ \mathtt{a}_1\mathtt{(e}_1\mathtt{)=a}_1\mathtt{(e}_2\mathtt{)} \wedge \ \ldots\ \wedge \mathtt{a}_k\mathtt{(e}_1\mathtt{)=a}_k\mathtt{(e}_2\mathtt{)} \Leftrightarrow \mathtt{e}_1\mathtt{=e}_2$$

  This formula expresses exactly the constraint that two entities are equal whenever their key attribute values are equal.

- **Cardinality constraints**, which are used to require minimal and maximal numbers of participations of entities in relationships (in specific roles), are given as (min, max) annotations to the lines representing the roles in the ERD.

  Assume a n-ary relationship type `R` between the entity types $\mathtt{E}_1$, ..., $\mathtt{E}_n$ and let the role in which $\mathtt{E}_1$ participates in `R` be annotated with the tuple (`min,max`). The semantics of this cardinality constraint is then expressed by the formula

  $$\begin{array}{l}\forall \mathtt{e}_1\mathtt{:E}_1\mathtt{.}\exists \mathtt{s:Set(E}_1\times\ldots\times\mathtt{E}_n\mathtt{).}\forall \mathtt{e}_2\mathtt{:E}_2\mathtt{,}\ldots\mathtt{,e}_n\mathtt{:E}_n\mathtt{.}\\ \quad \mathtt{((e}_1\mathtt{,}\ldots\mathtt{,e}_n\mathtt{)}\in\mathtt{s=true} \Leftrightarrow \mathtt{R(e}_1\mathtt{,}\ldots\mathtt{,e}_n\mathtt{)=true)} \wedge\\ \quad \mathtt{min}\leq\mathtt{card(s)=true} \wedge \mathtt{card(s)}\leq\mathtt{max=true;}\end{array}$$

  If the `min` component of the tuple is 0 or the `max` component is $*$, this formula can of course be simplified accordingly.

- **General constraints**, i.e. all static integrity constraint except keys and cardinalities, are already expressed as MINI-SPECTRUM formulae over the generated signature. They can therefore simply be taken over into the properties part of the specification.

3.5.2 Motivation of Semantics Definition

In our approach we assign a formal semantics to the entity-relationship model by assigning a specification of an abstract data type to each entity-relationship schema. In the following we will give a short explanation on why this way of semantics definition is sensible.

Informally, we can understand an entity-relationship schema as the definition of an abstract entity-relationship database. The schema then defines

1. the structure of the database (entities, attributes, relationships), and
2. integrity constraints which every state of the database has to obey.

This means that the schema can be seen as the definition of a set of allowed states of a (hypothetic) entity-relationship database.

The generated specification, on the other hand, is given its semantics as a set of Σ-*algebras*. These are algebras having a structure which represent the specification's signature (which itself reflects the structure of the data in terms of entity types, relationship types and attributes). The set of all algebras, which fulfill the properties demanded in the properties part of the specification[14], form the semantics of the specification. The formulae in the properties part reflect exactly the static integrity constraints.

It is therefore obvious that the model class of the generated specification represents the set of all allowed states of the entity-relationship database mentioned above. There is a bijection between the allowed database states described by the entity-relationship schema and the model class of the generated MINI-SPECTRUM specification.

3.5.3 Example

The following specification is generated from the example schema given in Section 3.4.4 according to the above rules and thus defines the formal semantics of this schema. It is hierarchically based on some auxiliary specifications (`Opt`, `Set`) and on the specifications defining the attribute sorts. This example is meant to illustrate the translation rules given above. More detailed information can be found in [Het95].

```
BIS = [ enriches Opt + Set + Nat + String + Date + Adr + Isbn;

-- Sorts representing entity types
sort Author, Keyword, Publication, Publisher;
```

[14]This set is often called the *model class* of the specification.

```
-- Selector functions representing attributes
Name        :  Author ⟶ String;
Dateofbirth :  Author ⟶ Date;
Address     :  Author ⟶ Adr;
Notion      :  Keyword ⟶ String;
Description :  Keyword ⟶ String;
Title       :  Publication ⟶ String;
Year        :  Publication ⟶ Nat;
ISBN        :  Publication ⟶ Opt Isbn;
P_Name      :  Publisher ⟶ String;
P_Address   :  Publisher ⟶ Adr;

-- Predicates representing relationship types
cites     :  Publication × Publication ⟶ Bool;
publishes :  Publisher × Publication ⟶ Bool;
uses      :  Publication × Keyword ⟶ Bool;
writes    :  Author × Publication ⟶ Bool;

axioms
-- Key constraints
∀a1,a2:Author.  Name(a1)=Name(a2) ∧
                Dateofbirth(a1)=Dateofbirth(a2) ⇒ a1=a2;
∀k1,k2:Keyword.  Notion(k1)=Notion(k2) ⇒ k1=k2;
∀p1,p2:Publication.  Title(p1)=Title(p2) ∧ Year(p1)=Year(p2) ⇒ p1=p2;
∀p1,p2:Publisher.  P_Name(p1)=P_Name(p2) ⇒ p1=p2;

-- Cardinality constraints
∀a:Author.∃s:Set(Author×Publication).∀p:Publication.
  ((a,p)∈s=true ⇔ writes(a,p)=true) ∧ 1≤card(s)=true;
∀k:Keyword.∃s:Set(Publication×Keyword).∀p:Publication.
  ((p,k)∈s=true ⇔ uses(p,k)=true) ∧ 1≤card(s)=true;
∀p:Publication.∃s:Set(Publication×Publisher).∀pp:Publisher.
  ((pp,p)∈s=true ⇔ publishes(pp,p)=true) ∧ 1=card(s);
∀p:Publication.∃s:Set(Author×Publication).∀a:Author.
  ((a,p)∈s=true ⇔ writes(a,p)=true) ∧ 1≤card(s)=true;
∀pp:Publisher.∃s:Set(Publisher×Publication).∀p:Publication.
  ((pp,p)∈s=true ⇔ publishes(pp,p)=true) ∧ 1≤card(s)=true;

-- Additional constraints
∀pp:Publisher,p:Publication.
publishes(pp,p)=true ⇒ ISBN(p)=NULL ∨ consistent(P_Name(pp),ISBN(p))=true;
endaxioms;
]
```

4 Mapping to System Model

In Section 1.1 we explained that all description techniques available in SysLab are formally based on the SysLab system model. In Sections 2.3 and 3.5, however, the algebraic specification language Spectrum is used for (indirectly) describing the formal semantics of those the introduced description techniques. This is possible because of the fact that the notion of Σ-Algebras, which is used in the semantics definition of Spectrum, is also present in the SysLab system model (cf. [GKR96]). This situation is depicted in Figure 2.

The semantics of the specification language Spectrum is defined algebraically, which means that specifications are interpreted by Σ-Algebras.

The semantics of Mini-Spectrum is given by a translation σ from Mini-Spectrum specifications to Spectrum specifications. The semantics of a Mini-Spectrum specification SP ($\mathcal{MOD}_{minispec}[SP]$) is defined to be semantics of the corresponding Spectrum specification ($\mathcal{MOD}_{spec}[\sigma[SP]]$).

EER diagrams are understood as a graphical notation for a special kind of Mini-Spectrum specifications, which means that there is a translation μ from EER diagrams to Mini-Spectrum specifications. The semantics of an EERD E is then defined to be the semantics $\mathcal{MOD}_{minispec}[\mu[E]]$ of the corresponding Mini-Spectrum specification $\mu[E]$.

Figure 2 explains that it is justified to use Σ-Algebras for defining the semantics of the introduced description techniques. It does, however, not explain how those description techniques contribute to the description of a system in the sense of the SysLab system model. This topic will be dealt with in the next section.

5 Describing the State of System Components

As described in [RKB95] and [GKR96], a system in SysLab consists of a set of hierarchically structured, communicating components. Data appear in this system view in two different contexts:

- Data are communicated as *messages* between system components.
- Data are stored in components as *component states*.

In the following we will deal with the question how the description techniques introduced in this paper can be applied for specifying the types of messages and the structure of component states.

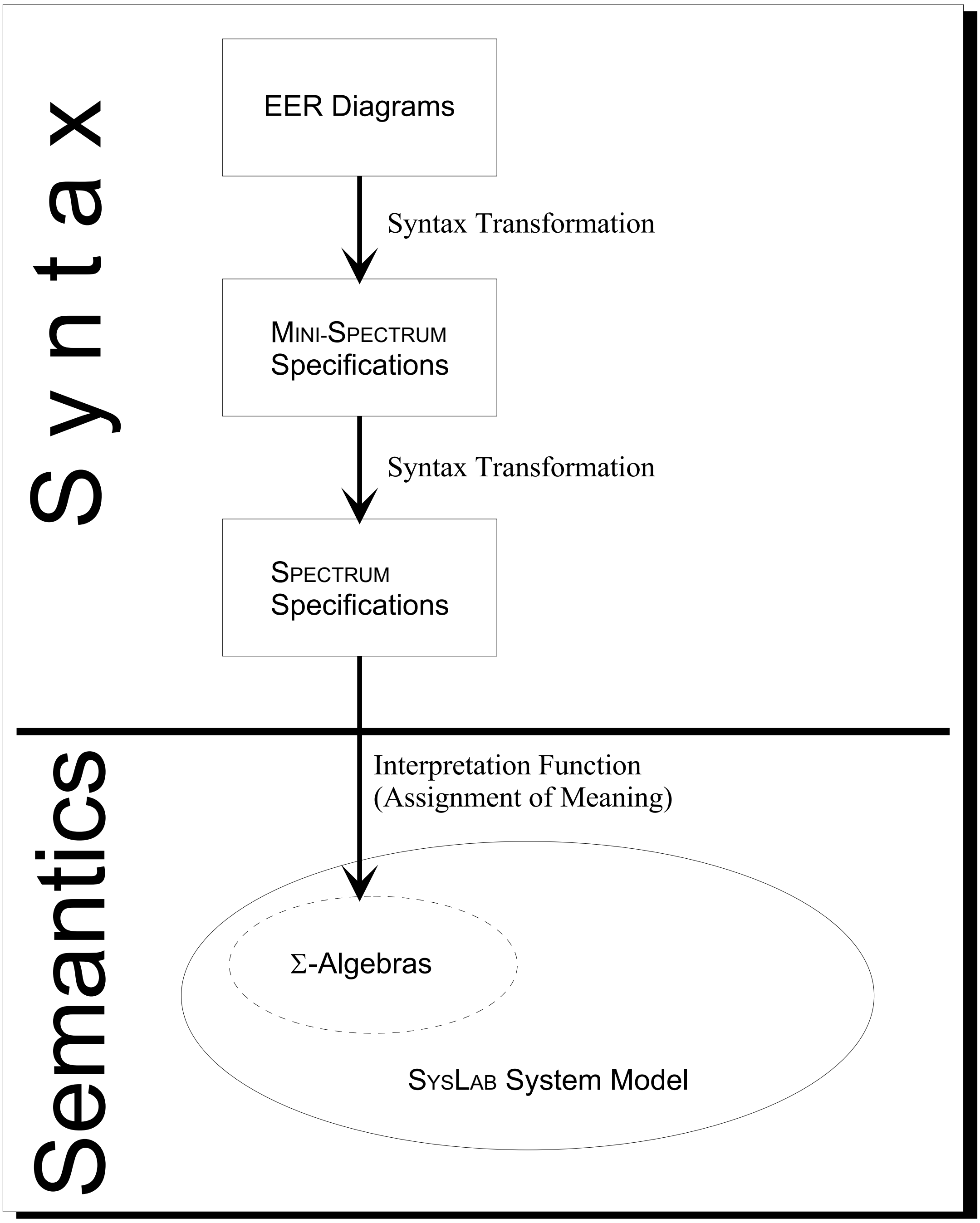


Figure 2: Mapping of Description Techniques to System Model

5.1 Messages

The type of messages sent over a communication channel is defined by associating a sort with the channel. The sort is provided by one of the Σ-Algebras contained in the system specification[15] (cf. [GKR96, Section 6.7]). The assignment of channels with sorts is done in the description technique for component classes which is not yet fixed.

5.2 Component States

In [GKR96, Section 6.6] the state of a basic[16] component is defined to be a triple consisting of an input message buffer, an output message buffer, and a data state:

$$State_c = (In_c \rightarrow D^*) \times Data_c \times (Out_c \rightarrow D^*)$$

The two message buffers are used in the system model to code the control state of the component. The only part of a component state, which a user of SysLab method wants to specify, is the data state $Data_c$. [GKR96, Section 6.7] states that this data state is associated with a sort provided by a Σ-Algebra.

The situation with data states is, however, not as simple as with messages. The data state of a component is likely to be a complex unit of information consisting of several parts[17]. We can distinguish two different kinds of data fields:

Simple Fields contain data elements of a sort provided by the specified ADTs. A simple field is thus associated with a sort contained in one of the existing Σ-Algebras.

Database Fields are used to store mass data described using EER diagrams. The state space described by an EERD, however, is given according to Section 3.5 as the semantics of a Mini-Spectrum specification associated with the diagram. This means that the state space is a set of Σ-Algebras. In order to associate the state space with a database field an interpretation of this state space in the form of a sort (*database sort*) is needed. There is a formal transition between those two representations of the state space, which is defined and analysed in [Het95]. We will not present the details of this transition (which in [Het95] is called *internalization*) here. The result

[15]Note that these Σ-Algebras can be the interpretation of Mini-Spectrum specifications as well as of EER diagrams. This means that entities can be communicated over channels.

[16]The state of a hierarchically decomposed component is completely determined by the states of its subcomponents and its communication medium.

[17]These parts of the data state will in the following be called *fields*. We avoid the term *component* here in order to not confuse these data fields with system components.

of this transition is a specification of a database sort which describes the set of all possible (according to the EERD) database states. A database state contains a set of entities for each entity type and a set of tuples of entities for each relationship type. Selector functions for those sets are provided. Of course, the specification allows to distinguish between database states, which are statically integer, and 'forbidden databases states, which violate static integrity. The example of Section 5.3 is meant to give an impression of what the specification of the database sort looks like.

⟨datastate⟩	::=	**datastate(** ⟨component⟩ **)** = [{⟨field⟩ // ,}$^+$]
⟨field⟩	::=	⟨id⟩ **:** {⟨prim-sort⟩ | ⟨db-sort⟩}

Figure 3: Description Technique for Data State of System Components

Figure 3 gives the syntax of a description technique which allows to specify the fields of a data state and to associate those fields with sorts. A data state specification of the form

```
datastate(c) = [ field1 :  sort1,
                          ⋮
                 fieldn :  sortn ]
```

is interpreted as MINI-SPECTRUM specification of a record sort which is, according to [GKR96], associated with the data state $Data_c$ of a component c:

```
c_state = [ enriches ...
  data cstate = mkcstate(field1 :  sort1, ..., fieldn :  sortn);
]
```

The sort `cstate` then is the sort which is assigned to the data state $Data_c$ of the basic component c.

5.3 Example

As an example we assume that we implement the bibliographic information system mentioned in the previous sections as a system built around one central database component. The state of this database component is therefore structured according to the EERM shown in Figure 1. With the notation defined above we can express this as:

```
datastate(DB-Component) = [ Database :  BIS ]
```

Informally, this statement says that the database component `DB-Component` consists of one field (a database field), which is described by the entity-relationship diagram identified by 'BIS'.

The formal meaning of the statement is defined by the following Mini-Spectrum specifications:

```
DB-Component_state = [ enriches BIS;
data DB-Componentstate = mkstate(Database :  BIS);
]
```

```
BIS = [ enriches Opt + Set + Nat + String + Date + Adr + Isbn;

-- Sorts representing entity types
sort Author, Keyword, Publication, Publisher;

-- Selector functions representing attributes
Name        :  Author ⟶ String;
Dateofbirth :  Author ⟶ Date;
Address     :  Author ⟶ Adr;
Notion      :  Keyword ⟶ String;
Description :  Keyword ⟶ String;
Title       :  Publication ⟶ String;
Year        :  Publication ⟶ Nat;
ISBN        :  Publication ⟶ Opt Isbn;
P_Name      :  Publisher ⟶ String;
P_Address   :  Publisher ⟶ Adr;

-- Sort representing database structure
data BIS_s = mkBIS_s [sel'Author :  Set Author,
                     sel'Keyword :  Set Keyword,
                     sel'Publication :  Set Publication,
                     sel'Publisher :  Set Publisher,
                     cites :  Set (Publication × Publication),
                     publishes :  Set (Publisher × Publication),
                     uses :  Set (Publication × Keyword),
                     writes :  Set (Author × Publication)];

-- Predicate to check static integrity
OK : BIS_s ⟶ Bool;

-- Database sort (only statically integer states)
sort BIS;

BIS_s2BIS : BIS_s ⟶ BIS;
```

```
BIS2BIS_s :  BIS ⟶ BIS_s;

BIS generated by BIS_s2BIS;

axioms ∀ B : BIS_s in
OK(B) ⇔
-- Referential integrity
  (∀p1,p2:Publication.(p1,p2)∈cites(B)=true ⇔
    p1∈sel'Publication(B)=true ∧ p2∈sel'Publication(B)=true) ∧
  (∀p1:Publisher,p2:Publication.(p1,p2)∈publishes(B)=true ⇔
    p1∈sel'Publisher(B)=true ∧ p2∈sel'Publication(B)=true) ∧
  (∀p:Publisher,k:Keyword.(p,k)∈uses(B)=true ⇔
    p∈sel'Publisher(B)=true ∧ k∈sel'Keyword(B)=true) ∧
  (∀a:Author,p:Publication.(a,p)∈writes(B)=true ⇔
    a∈sel'Author(B)=true ∧ p∈sel'Publication(B)=true) ∧
-- Key constraints
  (∀a1,a2:Author.a1∈sel'Author(B)=true ∧ a2∈sel'Author(B)=true ⇒
    Name(a1)=Name(a2) ∧ Dateofbirth(a1)=Dateofbirth(a2) ⇒ a1=a2) ∧
  (∀k1,k2:Keyword.k1∈sel'Keyword(B)=true ∧ k2∈sel'Keyword(B)=true ⇒
    Notion(k1)=Notion(k2) ⇒ k1=k2) ∧
  (∀p1,p2:Publication.p1∈sel'Publication(B)=true ∧ p2∈sel'Publication(B)=true ⇒
    Title(p1)=Title(p2) ∧ Year(p1)=Year(p2) ⇒ p1=p2) ∧
  (∀p1,p2:Publisher.p1∈sel'Publisher(B)=true ∧ p2∈sel'Publisher(B)=true ⇒
    P_Name(p1)=P_Name(p2) ⇒ p1=p2) ∧
-- Cardinality constraints
  (∀a:Author.a∈sel'Author(B)=true ⇒
    (∃s:Set(Author×Publication).∀p:Publication.p∈sel'Publication(B)=true ⇒
      ((a,p)∈s=true ⇔ (a,p)∈writes(B)=true) ∧ 1≤card(s)=true)) ∧
  (∀k:Keyword.k∈sel'Keyword(B)=true ⇒
    (∃s:Set(Publication×Keyword).∀p:Publication.p∈sel'Publication(B)=true ⇒
      ((p,k)∈s=true ⇔ (p,k)∈uses(B)=true) ∧ 1≤card(s)=true)) ∧
  (∀p:Publication.p∈sel'Publication(B)=true ⇒
    (∃s:Set(Publication×Publisher).∀pp:Publisher.pp∈sel'Publisher(B)=true ⇒
      ((pp,p)∈s=true ⇔ (pp,p)∈publishes(B)=true) ∧ 1=card(s))) ∧
  (∀p:Publication.p∈sel'Publication(B)=true ⇒
    (∃s:Set(Author×Publication).∀a:Author.a∈sel'Author(B)=true ⇒
      ((a,p)∈s=true ⇔ (a,p)∈writes(B)=true) ∧ 1≤card(s)=true)) ∧
  (∀pp:Publisher.pp∈sel'Publisher(B)=true ⇒
    (∃s:Set(Publisher×Publication).∀p:Publication.p∈sel'Publication(B)=true ⇒
      ((pp,p)∈s=true ⇔ (pp,p)∈publishes(B)=true) ∧ 1≤card(s)=true)) ∧
-- Additional constraints
  (∀pp:Publisher,p:Publication.
    pp∈sel'Publisher(B)=true ∧ p∈sel'Publication(B)=true ⇒
    ((pp,p)∈publishes(B)=true ⇒
      ISBN(p)=NULL ∨ consistent(P_Name(pp),ISBN(p))=true));
```

```
BIS_s2BIS(B) ≠ ⊥ ⇔ OK(b)=true;
OK(b)=true ⇒ BIS2BIS_s(BIS_s2BIS(B))=B;
endaxioms;
]
```

Remark In the specification given above the sort BIS is introduced in two steps. First, a sort BIS_s defines the structure of the database without considering static integrity. The sort BIS is thereafter defined as a restriction of BIS_s such that it comprises exactly the statically integer database states, which are distinguished by the OK predicate.

6 Conclusion

In the paper at hand we have presented description techniques for data in the SysLab method, thus covering the method's data oriented system view. Following the basic principle of the SysLab project, these description techniques were given a formal semantics and were related to the SysLab system model. The non-data oriented system views of the SysLab method are currently being dealt with in the SysLab project in a similar way.

From the point of view of data there is another interesting aspect in modeling information systems, which has not been dealt with in this paper. It is the treatment of *dynamic integrity constraints*. They can be used to specify the system behaviour from a data oriented point of view. In the object-oriented method Fusion [CAB+94], for example, they appear in the form of so-called *lifecycles*. In SSADM [DCC92], the notion of *entity life histories* deals with dynamic data integrity. In the context of the SysLab project this is an interesting topic for further research.

References

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus — Revised Version. Technical Report TUM-I9202-2, Technische Universität München, Institut für Informatik, 1993.

[BFG+93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language Spectrum. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.

[BFG+93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development — The Fusion Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Che76] P. Chen. The entity-relationship model — toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

[DCC92] Ed Downs, Peter Clare, and Ian Coe. *Structured Systems Analysis and Design Method — Application and Context*. Prentice Hall, 1992.

[GKR96] R. Grosu, C. Klein, and B. Rumpe. Enhancing the SYSLAB System Model with State. Technical Report TUM-I9631, Technische Universität München, Institut für Informatik, 1996.

[GN94] Radu Grosu and Dieter Nazareth. The Specification Language SPECTRUM - Core Language Report V1.0. TUM-I 9429, Technische Universität München, 1994.

[Gog94] M. Gogolla. *An Extended Entity-Relationship Model. Fundamentals and Pragmatics*, volume 767 of *Lecture Notes in Computer Science*. Springer, 1994.

[GR94] R. Grosu and F. Regensburger. The Logical Framework of SPECTRUM. Technical Report TUM-I9402, Technische Universität München, 1994.

[GTWW75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. G. Wright. Abstract Data-Types as Initial Algebras and Correctness of Data Representations. In *Proceedings Conference on Computer Graphics, Pattern Recognition and Data Structure*, 1975.

[Gut75] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, Department of Computer Science, University of Toronto, 1975.

[Het95] R. Hettler. *Entity/Relationship Datenmodellierung in axiomatischen Spezifikationssprachen*. TECTUM, Marburg, 1995.

[HMM86] R.W. Harper, D.B. MacQueen, and R.G. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Univ. Edinburgh, 1986.

[Hoh93] U. Hohenstein. *Formale Semantik eines erweiterten Entity-Relationship-Modells*, volume 4 of *Teubner-Texte zur Informatik*. Teubner, 1993.

[JRP91a] M. B. Josephs and D. Redmond-Pyle. Entity-Relationship Models Expressed in Z: A Synthesis of Structured and Formal Methods. Technical Report PRG-TR-20-91, Oxford University Programming Research Group, 1991.

[JRP91b] M. B. Josephs and D. Redmond-Pyle. A Library of Z Schemas for use in Entity-Relationship Modelling. Technical Report PRG-TR-21-91, Oxford University Programming Research Group, 1991.

[Pae95] B. Paech. A Methodology Integrating Formal and Informal Software Development. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods in Software Engineering Practice*, pages 61–68, Seattle, Washington, 1995.

[RKB95] B. Rumpe, C. Klein, and M. Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab Systemmodell -. Technical Report TUM-I9510, Technische Universität München, Institut für Informatik, March 1995.

[SS83] G. Schlageter and W. Stucky. *Datenbanksysteme: Konzepte und Modelle*. Teubner, Stuttgart, 1983.