LEX: A Case Study in Development and Validation of Formal Specifications [1]

Andreas Ramses Heckler [a], Rudolf Hettler [b],
Heinrich Hußmann [c], Jacques Loeckx [a], Wolfgang Reif [d],
Gerhard Schellhorn [d] and Kurt Stenzel [d]

[a] *Universität Saarbrücken / Fachbereich 14 / Postfach 15 11 50/ D–66041 Saarbrücken*

[b] *Forschungsinstitut für Angewandte Software–Technologie e. V. (FAST) / Arabellastraße 17 / D–81925 München*

[c] *Siemens AG / Bereich Öffentliche Kommunikationsnetze / Hofmannstr.51 / D–81359 München*

[d] *Universität Ulm / Fakultät für Informatik / Abt. Programmiermethodik / Oberer Eselsberg / D–89069 Ulm*

Abstract

The paper describes an experiment in the combined use of various tools for the development and validation of formal specifications. The first tool consists of a very abstract, (non–executable) axiomatic specification language. The second tool consists of an (executable) constructive specification language together with a specification environment. Finally, the third tool is a verification system. The first two tools were used to develop two specifications for the same case study, viz. a generic scanner similar to the tool Lex present in Unix. Reflecting the nature of the tools the first specification is abstract and non–executable, whereas the second specification is less abstract but executable. Thereupon the verification system was used to formally prove that the second specification is consistent with the first one in that it describes the same problem. During this proof it appeared that both specifications contained conceptual errors ("adequacy errors"). It is argued that the combined use of tools similar to those employed in the experiment may substantially increase the quality of software.

[1] This work has been supported by the German ministry of research and technology (BMFT), as part of the compound project KORSO (Korrekte Software).

1 Introduction

1.1 Correctness and Adequacy of Formal Specifications

To increase the reliability of software it has been proposed to start the design of a program by drawing up a formal — and hence precise — specification of the problem to be solved. Taking this specification as a basis one then develops — in one or several steps — the wanted program. Finally, in a so-called verification step one formally proves — by hand or by using a verification system — that the program obtained fulfills the specification. Meanwhile different refinements and variants of this method have been proposed. The main point to be stressed here is that the verification step of the method guarantees that the program is *correct* with respect to the specification, i.e. that it is free of "programming errors".

Unfortunately, even a verified program may fail to solve the given problem. This occurs when the specification does not correctly describe the problem. Following a notion introduced, for instance, in [5] such a conceptual error in (the design of) a specification is called an *adequacy error*.

It is important to note that adequacy can only be tested, not proved. The reason is that adequacy relates a formal notion (viz. a specification) to an informal one (viz. a "problem"). Clearly, a negative result of an adequacy test disproves the adequacy while a positive result merely may increase the confidence in the adequacy.

A classical test method for detecting inadequacies in a specification is rapid prototyping. Of course, this method requires the specification to be executable.

In the present paper we propose a more elaborate method for testing the adequacy of a specification and we illustrate its use in a case study. The method — called *method of complementary specification* — is presented in Section 1.2. The case study is shortly described in Section 1.3.

1.2 The Method of Complementary Specification

As indicated above the goal of the method is to perform an adequacy test of specifications. It consists of five steps:

(i) The problem to be specified is given an informal description; this description has to be as precise as possible and may contain formal parts.

(ii) An axiomatic specification for the problem is drawn up. This specification should keep as close to the informal description of step (i) as possible. Moreover, the specification should be abstract in the sense that it should avoid any overspecification. To this end it may make use of, for instance, the full power of first-order logic. The resulting specification will be loose

in that it may possess different (i.e. non-isomorphic) models. Henceforth, we refer to this specification as the *requirement specification*.

(*iii*) Independently from step (*ii*) but based on its result a specification is drawn up where the special form of the axioms ensures executability and — at the price of some proof effort — termination ("constructive specification"). The specifier has to take into consideration the informal description of step (*i*) as well as the formal requirement specification of step (*ii*). Again, the goal is to obtain a formal specification but now with axioms of a particular form. The specifier should abstract from any consideration concerning the efficiency of the execution. Being executable this specification has only one model (up to isomorphism). Henceforth we refer to this specification as the *design specification*.

(*iv*) The design specification may be used to perform some rapid prototyping and thus already detect some inadequacies. This step is does not belong to the method proper.

(*v*) It is formally proved that the model of the design specification is a model of the requirement specification.

The essential step of the method is, of course, step (*v*). Note that a positive result of this verification merely proves that both specifications are "mutually consistent" im[lying, in particular, that the requirement specification has a model. It fails to prove the adequacy of the specifications in case they both contain exactly the same adequacy errors. We argue that the probability of this case is small because the two specifications are supposed to have been developed independently from each other and, more importantly, because the development of requirement specifications is based on axiomatic thinking whereas the development of design specifications is based on algorithmic thinking.

The method was applied on the case study to be described in Section 1.3. In this experiment the steps (*ii*) and (*iii*) were performed by different teams: step (*ii*), which is described in Section 3, was performed at the Technische Universität München and step (*iii*) (Section 4) at the Universität Saarbrücken. Step (*v*) that is documented in Section 5 was performed with the help of an automatic verification system at the Universität Karlsruhe. Note that neither step (*i*) nor step (*iv*) were carried out in this case study. As step (*iv*) is facultative, it can be safely left out. Step (*i*) could be omitted because of the very specific nature of the case study. The theory of lexical analysis is a well-known subject described in many publications such as [1]. Furthermore, the existing UNIX tool LEX constituted a good model that was tried out in case of doubt. Together, [1] and LEX provide more information about the problem to be solved than is usually contained in an informal problem description.

It is interesting to note that the method proposed may be integrated into a general method for the design of reliable software. More precisely, the design specification may be considered to be the first refinement step in the development of a program starting from the (requirement) specification.

1.3 The Case Study Lex

The software problem of the case study consists in the construction of a lexical analyzer similar to the Unix tool Lex. Of course, the case study has an academic flavor as there exist efficient and manifestly adequate implementations of the problem. On the other hand the case study is of moderate size and well-suited to illustrate the method described proposed.

Informally, lexical analysis — also known as scanning — consists in breaking up a string into substrings according to a set of regular expressions. More precisely, each substring has to belong to the language defined by a regular expression of the set. The result is the list of regular expressions corresponding to the substrings. Hence lexical analysis may in principle be modeled by a function, say `scan`, taking two arguments. The first argument is a string, viz. the string to be analyzed, the second argument is a set of regular expressions and the function value is a list of regular expressions.

Actually, this definition of the function `scan` lacks precision. In fact, depending on the string and the set of regular expressions the problem of lexical analysis may have several solutions or, alternatively, no solution. This difficulty will be addressed in Section 3.5.

By the way, for efficiency reasons the actual Unix tool Lex constitutes a somewhat peculiar implementation of the function `scan`. In a "preprocessing phase" the tool turns the second argument of `scan` (viz. a set of regular expressions) into a program called scanner. In a subsequent "run phase" this program reads in the first argument of the function `scan` and yields the function value as a result. For the user of the Unix tool Lex this scanner-generating feature is important as it improves the flexibility and efficiency of the tool. In the specification of the problem this feature may be ignored. Actually, it may be viewed as resulting from a decision taken during the design of an implementation of `scan`, i.e. during the development of a program for the specification. Hence this problem is outside the scope of the present paper.

2 Tools Used for the Experiment

2.1 Spectrum

The Spectrum project at the Technische Universität München concentrates on the process of developing precise and well-structured specifications on an abstract level. Spectrum comprises a specification language, a deduction calculus and a development methodology. As the name of the project suggests Spectrum is intended to encompass a wide range of specification styles. For instance, the constructive specification language Obscure (which was used

in the case study for step (*iii*) of Section 1.2) can be viewed semantically as a sublanguage of SPECTRUM.

SPECTRUM is based on classical algebraic specification techniques. However, in contrast to most algebraic specification languages it explicitly supports the use of partial functions. Moreover, SPECTRUM is not restricted to equational or conditional axioms but provides full first-order predicate logic extended by some second-order principles.

SPECTRUM is oriented towards the development of functional programs. A number of concepts have therefore been taken over from functional programming languages such as parametric polymorphism and higher-order functions.

The SPECTRUM specifications contained in Section 3 and 4 are provided with comments. For the reader acquainted with some classical concepts and notation from algebraic specification techniques these comments should suffice to enable the reading of the paper. For more information on SPECTRUM the reader may consult [3,4].

2.2 OBSCURE

The *specification language* OBSCURE is described in [10]. It has been designed to be a simple but robust tool. While allowing operator overloading it provides neither polymorphism nor higher-order functions. On the other hand, stringent context conditions allow to automatically generate formulas that express the persistency of a specification. Note that OBSCURE is a language scheme rather than a language because it does in particular not fix the specification method used to draw up the "elementary" specifications i.e. used for specification-in-the-small.

The *specification environment* OBSCURE is described in [13]. It consists, among others, of an editor, a data base, a parser performing a complete syntactical check and facilities for rapid prototyping. The specifications are written in the specification language OBSCURE instantiated with a constructive specification method, viz. the algorithmic specification method [11]. The use of this particular specification method makes OBSCURE specifications look like programs written in a very abstract programming language.

As already indicated, specifications written in the specification language OBSCURE and, especially, specifications written with the help of the environment OBSCURE may be viewed as (syntactical variants of) SPECTRUM specifications. In order not to bother the reader with additional syntactical details the OBSCURE specifications in Section 4 are written in SPECTRUM notation.

2.3 KIV

KIV stands for *Karlsruhe Interactive Verifier* and is an advanced support tool

for correct software development for large sequential systems [15,17,6,7]. It supports the entire design process from formal specification to verified executable code and contributes to an economically applicable verification technology. Substantial verification has been done using KIV. The current productivity is between 1000 to 2000 lines of verified code per year.

KIV relies on an ASL-style [19], first-order algebraic specification language to describe hierarchically structured software systems. Specification components are implemented by stepwise refinement using (functional) program modules. The specifier has to follow a strict decompositional design discipline leading to modular systems with compositional correctness. As a consequence the verification effort for a modular system becomes linear in the number of its modules. KIV offers a powerful interactive verification component for module correctness based on proof tactics. It combines a high degree of automation with an elaborate interactive proof engineering environment.

A correctness management of KIV keeps track of the development graph (visualizing the development process), proof obligations and proofs. Furthermore, it computes and visualizes the impact of modifications on the correctness of other components. An interesting feature of the KIV verification methodology is the tight coupling of error detection, correction (to specifications or programs) and an intelligent reuse of proofs. Actually, KIV offers a mechanism that goes far beyond proof replay [18].

3 A Requirement Specification for LEX

This section presents a requirement specification for the scanner described in Section 1.3 (see also [8]). This specification is written in SPECTRUM and was developed at the Technische Universität München as the result of step (*ii*) of the method of complementary specification. For a detailed description of the specification language SPECTRUM and its standard library on which this specification is based, the reader is referred to [3,4].

The specification is presented in a bottom–up manner, starting from "elementary" specifications and ending up with a specification of the function `scan` performing the lexical analysis.

3.1 Naturals, Lists, Characters and Strings: the Elementary Specifications

The Standard Library of SPECTRUM [4] contains, among others, the specifications of lists, natural numbers, characters and character strings.

To provide a flavor of SPECTRUM Figure 1 presents the specification called `LISTS` that introduces polymorphic lists together with some usual list operations. The following remarks should help to understand this specification (and other SPECTRUM specifications presented in this paper as well):

```
LISTS = { -- polymorphic lists
   data List α = [] | cons(!first: α,!rest: List α);
   List::(EQ)EQ;

   .++.:List α × List α → List α prio 10:left;
   .++. strict total;

   axioms ∀ s,s':List α, e:α in
   {l1}    [] ++ s = s;
   {l2}    cons(e,s') ++ s = cons(e,s' ++ s);
   endaxioms;
}
```

Fig. 1. A SPECTRUM specification of polymorphic lists

- The text to the right of `--` is a comment.
- SPECTRUM's `data` construct is similar to the construct `data` or `datatype` in functional languages like ML [14] or Haskell [9]. It introduces a new sort (in the case of the example of Figure 1 the polymorphic sort `List` α) together with its constructors ([] and `cons`) and selectors (`first`, `rest`). The exclamation marks in front of the selectors may be ignored in a first reading: they merely express the strictness of the constructor in the respective arguments.
- SPECTRUM provides a notion of sort classes very similar to the type classes of Haskell. In Figure 1, the line `List::(EQ)EQ` expresses that whenever the sort constructor `List` is applied to a sort from the (previously defined) class `EQ`, it yields a sort which again is from `EQ`. Note that independently from this property `List` may also be applied to arbitrary sorts.
- Functions are defined syntactically by signatures and semantically by axioms. In Figure 1, the line

  `.++. : List` $\alpha \times$ `List` $\alpha \rightarrow$ `List` α `prio 10:left`

  introduces the infix function `.++.` which is left associative and has a certain binding priority. Semantically this function is defined by three axioms. Two of them are given as logical formulae embraced by the keywords `axioms` and `endaxioms`. Such logical axioms can be given names (in our case `l1` and `l2`) embraced by curly brackets in front of the axioms. The third axiom is given by the line `.++. strict total` and requires the function `.++.` to be strict and total.
- According to the specification of Figure 1 the constructors of the sort `List` α are [] and `cons`. Writing down concrete lists with the help of these constructors is quite cumbersome. As lists are an important specification concept SPECTRUM provides a shorthand notation: a finite list may be denoted by enumerating its elements. For instance, `[a,b,c,d]` denotes the list consisting of the elements `a`, `b`, `c` and `d`.

In addition to lists the Standard Library of SPECTRUM contains a specifica-

tion `NATURALS` introducing the sort `Nat` together with the usual operations on natural numbers. A further specification is `CHARACTERS` introducing the sort `Char`. Finally, the specification `STRINGS` introduces the sort `String` of character strings. This sort is defined as an instance of the sort `List` α, viz. as

```
String = List Char;
```

As strings constitute a special case of lists, the shorthand notation for lists mentioned above is applicable to strings, too. As this shorthand notation is still lengthy, SPECTRUM provides an additional shorthand notation for strings: ¨¨ stands for [ ] and, for instance, ¨`abcd`¨ for [`a,b,c,d`].

3.2 Some List Operations: the Specification `EXT_LISTS`

In the specification of the function `scan` the list operations provided by the standard specification `LISTS` of Figure 1 do not suffice. The specification `EXT_LISTS` (for "extended lists") in Figure 2 introduces the required additional list operations.

The meaning of the different functions introduced should be clear. Note that the function `flatten` maps lists of lists of elements into lists of elements by removing all parentheses but the outermost ones.

The specification of Figure 2 consists of two parts introducing the first three and the remaining four functions respectively. This presentation is made for clarity only and is semantically irrelevant.

3.3 The Syntax of Regular Expressions: the Specification `REGEXP`

Classically the set of all regular expressions for a given set of characters is defined inductively (see e.g. [1]) [2] :

- *(empty regular expression)* $\emptyset$ is a regular expression;
- *(empty word)* ε is a regular expression;
- *(atomic expression)* each character is a regular expression;
- *(sequence)* if r_1 and r_2 are regular expressions, then $(r_1 \circ r_2)$ is a regular expression;
- *(sum)* if r_1 and r_2 are regular expressions, then $(r_1 \parallel r_2)$ is a regular expression;
- *(Kleene–star)* if r is a regular expression, then $(** r)$ is a regular expression.

A SPECTRUM specification reflecting this definition is in Figure 3. The reader should not bother about the syntactical details of this definition. The only important point to remember is that the specification introduces a sort `Regexp`

[2] The use of the quite unusual symbols $\circ$,|| and ** for the operations on regular expressions stems from lexical restrictions imposed by the language SPECTRUM.

```
EXT_LISTS = {

enriches LISTS + NATURALS;

-- all functions in this specification are strict and total
strict total;

mklist  : α → List α;
length  : List α → Nat;
flatten : List (List α) → List α;

axioms ∀s:List α,ss:List(List α),e:α in
mklist(e) = cons(e,[]);
length([]) = 0;
length(cons(e,s)) = succ(length(s));
flatten([]) = [];
flatten(cons(s,ss)) = s ++ flatten(ss);
endaxioms;

.∈.           : α::EQ ⇒ α × List α → Bool  prio 6;
.is_prefix_of. : α::EQ ⇒ List α × List α → Bool prio 6;
.is_postfix_of.: α::EQ ⇒ List α × List α → Bool prio 6;
_precedes_in_  : α::EQ ⇒ α × α × List α → Bool prio 6;
--prio expresses the operator priority

axioms α::EQ ⇒ ∀s,s':List α
               ∀e,e':α
in
e ∈ s = ∃s1,s2. s = s1 ++ mklist(e) ++ s2;
s is_prefix_of s' = ∃s''. s' = s ++ s'';
s is_postfix_of s' = ∃s''. s' = s'' ++ s;
_precedes_in_(e, e', s) = ∃s1,s2,s3. ¬(e'∈ s1) ∧
    s = s1 ++ mklist(e) ++ s2 ++ mklist(e') ++ s3;
endaxioms;
}
```

Fig. 2. The specification EXT_LISTS

, the carrier set of which consists of the set of all regular expressions defined in the informal definition above. As a minor difference a regular expression consisting of a single character, say `c`, is written `mkreg(c)` rather than `c`.

3.4 Matching Strings with Regular Expressions: the Specification MATCH

The syntax of regular expressions was introduced in Section 3.3. The goal of the present section is to define their semantics.

```
REGEXP = {

enriches CHARACTERS;

data Regexp = ∅ | ε | mkreg (! Char)
        | .∘. (! Regexp,! Regexp) prio 11
        | .||. (! Regexp,! Regexp) prio 10
        |  ** (! Regexp);

Regexp :: EQ;

}
```

Fig. 3. The specification REGEXP

The meaning of a regular expression is a language, i.e. a set of strings. Classically, this language is defined inductively. Alternatively, one may introduce a relation, say `.matches.`, between the set of regular expressions and the set of all strings. Per definition, a regular expression r matches a string s , if and only if s belongs to the language defined by r . This relation may be defined by induction on the structure of regular expressions:

- $\emptyset$ does not match any string;
- ε only matches the empty string, viz. ¨¨;
- a regular expression `mkreg`(c) only matches the string `mklist`(c);
- a regular expression of the form $(r_1 \circ r_2)$ matches a string s, if and only if s may be broken up into two substrings, say s_1 and s_2 with $s = s_1{+}{+}s_2$, such that r_1 matches s_1 and r_2 matches s_2;
- a regular expression of the form $(r_1 \parallel r_2)$ matches a string s, if and only if either r_1 matches s or r_2 matches s (or both);
- a regular expression of the form $(** r)$ matches a string s, if and only if either s is the empty string or s may be broken up into a number of substrings each of which is matched by r.

This definition may be "translated" into SPECTRUM in a straightforward way. Note that apart from the function `.matches.` the specification MATCH (Figure 4) also introduces a function `.is_prefix_match_of.`. This function constitutes a shorthand notation for later use.

3.5 Lexical Analysis: the Specification SCAN

At last it is possible to specify the function `scan` already mentioned in Section 1.3 that performs the lexical analysis. Remember that the function takes a string and a set of regular expressions as arguments and has as its value a list of regular expressions. As already indicated the lexical analysis of a string with respect to a set of regular expressions may have several solutions. A

```
MATCH = {

enriches STRINGS + EXT_LISTS + REGEXP;

.matches.            : Regexp × String → Bool            prio 6;
.is_prefix_match_of. : (String × Regexp) × String → Bool prio 6;

.matches., .is_prefix_match_of. strict total;

axioms ∀ c:Char, r1,r2:Regexp, s,s':String in
¬(∅ matches s);
ε matches s = (s = "");
mkreg(c) matches s = (mklist(c) = s);
(r1∘r2) matches s = ∃s1,s2. s = s1 ++ s2 ∧ r1 matches s1 ∧
                            r2 matches s2;
(r1||r2) matches s = r1 matches s ∨ r2 matches s;
(**r1) matches s = (s = "" ∨ ∃ss: List String. s = flatten(ss) ∧
                   ∀s' : String. s' ∈ ss ⇒ r1 matches s');

(s,r1)is_prefix_match_of s' ⇔ r1 matches s ∧ s is_prefix_of s';
endaxioms;
}
```

Fig. 4. The specification MATCH

specification of the function scan would therefore be correspondingly loose. Instead, the authors have opted for a more restricted definition of the notion of lexical analysis and, correspondingly, of the function scan. Being inspired from the strategy applied in the UNIX tool LEX this restricted definition is characterized by five "design decisions", the first four of which are:

- (*i*) the second argument of the function scan is chosen to be a list of regular expressions rather than a set of regular expressions;
- (*ii*) the list of regular expressions constituting the value of the function scan is defined iteratively; at each iteration step one chooses the regular expression matching the longest possible prefix of the string that remains to be matched;
- (*iii*) in case several regular expressions match the same string one chooses the leftmost one in the list of regular expressions constituting the second argument of the function scan;
- (*iv*) in order to document an unsuccessful lexical analysis the value of the function scan is defined to be a pair; the first element of this pair is — as before — the list of the matching regular expressions; the second element is a suffix of the string to be matched, viz. the suffix that remains unmatched according to the design decision (*ii*).

An example illustrating the design decision (*ii*) is[3]:

$$\texttt{scan(¨abb¨ , [a , a} \circ \texttt{b , b]) = ([a} \circ \texttt{b , b], ¨¨)}$$

An example illustrating (*iii*) is:

$$\texttt{scan(¨a¨ , [a || b , a]) = ([a || b] , ¨¨)}$$

An example illustrating (*iv*) is:

$$\texttt{scan(¨aabb¨ , [a} \circ \texttt{a , b} \circ \texttt{b , a} \circ \texttt{a} \circ \texttt{b]) = ([a} \circ \texttt{a} \circ \texttt{b] , ¨b¨)}$$

By the way, without the design decision (*ii*) the lexical analysis of the last example could have been successful yielding [a ∘ a , b ∘ b] as a result!

Actually, even with the above design decisions the function `scan` is not completely defined. The reason is that the above design decisions do not exclude the repeated matching of an empty string. Three examples illustrating this fact are:

- possible values of `scan(¨a¨ , [a,`ε`])` are
  `([ a ] , ¨¨),`
  `([ a , `ε` ] , ¨¨),`
  `([ a , `ε` , `ε` ] , ¨¨),`
  ...
- possible values of `scan(¨b¨ , [a,`ε`])` are
  `( [] , ¨b¨),`
  `( [`ε`] , ¨b¨),`
  `( [`ε`,`ε`] , ¨b¨),`
  ...
- possible values of `scan(¨a¨ , [**a])` are
  `([**a] , ¨¨),`
  `([**a , **a], ¨¨),`
  `([**a , **a , **a] , ¨¨),`
  ...

While this looseness may easily be accounted for in a requirement specification the authors decided to stick to the solution implemented by the algorithm of the UNIX tool LEX. This leads to an additional "design decision" removing the looseness illustrated above:

(*v*) any match of the empty string is disregarded.

The specification of Figure 5 defines the function `scan`. In addition it introduces a function `is_longest_prefix_match_of` that implements the strategy of the longest prefix expressed by the design decision (*ii*).

The value of the function `scan` consists of a pair the components of which can be accessed by the selectors `tokens` and `unprocessed` of the composite data type `Scan_Result`.

[3] For reasons of readability we leave out the application of the constructor `mkreg` in the following equations.

```
SCAN = { enriches MATCH;

.is_prefix_match_of. : (String × Regexp) × String → Bool prio 6;
.is_prefix_match_of. strict total;

axioms ∀ r1,s,s' in
(s,r1)is_prefix_match_of s' ⇔ r1 matches s ∧ s is_prefix_of s';
endaxioms;

.is_longest_prefix_match_of. :
     (String × Regexp) × (String × List Regexp) → Bool prio 6;
.is_longest_prefix_match_of. strict total;

axioms ∀s,s',r,rs in
(s',r) is_longest_prefix_match_of (s,rs) ⇔
  r∈rs ∧ (s',r) is_prefix_match_of s ∧
  ∀s1,r1. r1∈rs ∧ (s1,r1) is_prefix_match_of s ∧ (s1,r1)≠(s',r) ⇒
                length(s1)<length(s') ∨ (length(s1)=length(s') ∧
                            _precedes_in_(r,r1,rs));
endaxioms;

data Scan_Result = mkres(! tokens: List Regexp,
                         ! unprocessed: String);

scan : String × List Regexp → Scan_Result;
scan strict total;

axioms ∀s,s',rs,ts in
scan(s,rs) = mkres(ts,s') ⇒
       s' is_postfix_of s ∧
       (∀r. r ∈ ts ⇒ r ∈ rs) ∧
       (ts = [] ⇒ s=s' ∧
              ∀s'',r. r ∈ rs ∧ s'' ≠ "" ⇒
                    ¬((s'',r) is_prefix_match_of s)
       ) ∧
       (ts ≠ [] ⇒ ∃s1,s2. s = s1 ++ s2 ∧
              scan(s2, rs) = mkres(rest(ts), s') ∧
              (s1,first(ts)) is_longest_prefix_match_of (s,rs)
       );
endaxioms;
}
```

Fig. 5. The specification SCAN

Note that scan is defined as a relation. That it is effectively a function or, equivalently, that its definition is consistent, is not apparent but is a result of the verification of Section 5.

4 A Design Specification for LEX in OBSCURE

According to Step (*iii*) of the method proposed the present section presents a design specification for the same case study. It was developed at the Universität Saarbrücken with the help of the OBSCURE system.

As a difference with the declarative and non-executable requirement specification the design specification is constructive and hence executable. On the other hand the general structure of the design specification should be close to that of the requirement specification in order to facilitate the verification of Step (*v*) of the method proposed. To this end the modularized structure of the design specification was chosen to be identical with that of the requirement specification (see Figure 6). Moreover, sorts and functions with the same (or with a similar) intended meaning have been given the the same (or similar) names. Finally, the axioms of the requirement specification that are already "constructive" have been "taken over". Hence, the design specification fundamentally differs from the requirement specification in those places where the axioms of the latter contain existential quantifier.

To facilitate the reading of the present paper the design specification is presented as a SPECTRUM specification. It constitutes a straightforward "translation" of the OBSCURE specification developed in Saarbrücken. The reader interested in the original version in OBSCURE may consult [2]. This paper also contains additional details on the — non-trivial! — algorithm implicitly specified by the axioms of the design specification.

This section does not present the complete design specification but describes the differences that distinguish the design from the requirement specification. The modularization structure of the design specification is chosen to be identical to that of the requirement specification. It is shown in Figure 6. Therefore, it is possible to regard the basic specifications (`SCAN`, `MATCH`, `REGEXP`, ...) one by one.

4.1 The Elementary Specifications

The elementary specifications used in the design specification are essentially the same as those used in the requirement specification and shortly described in Section 3.1. As a minor difference the design specification uses two additional list functions, viz.

$$\begin{array}{ll} \texttt{last} & : \texttt{List}\ \alpha \to \alpha \\ \texttt{allbutlast} & : \texttt{List}\ \alpha \to \texttt{List}\ \alpha \end{array}$$

These functions deliver the last element of a list and the list from which the last element is removed respectively. By the way, the functions `first` and `rest` of Section 3.1 are not used.

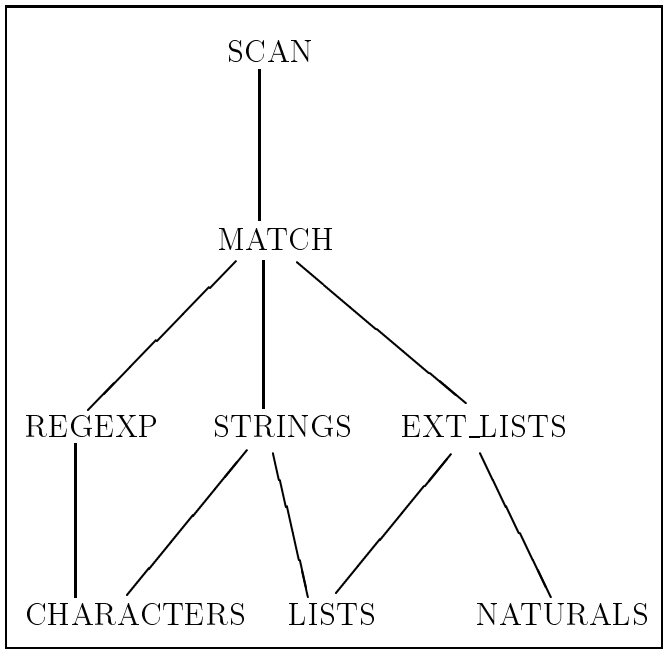

Fig. 6. Modularization structure of the case study

4.2 The Specification EXT_LISTS

This design specification differs from the requirement specification EXT_LISTS of Section 3.2 by the axioms for the functions .∈. and .is_prefix_of.. Moreover, as the functions .is_postfix_of. and _precedes_in_ are not used in the subsequent specifications their declaration and their axioms are missing.

The axioms for .∈. and .is_prefix_of. are:

```
¬(e ∈ []);
e ∈ cons(e,s);
e ≠ e' ⇒ (e ∈ cons(e',s)) = (e ∈ s);

[] is_prefix_of s;
¬ (cons(e,s) is_prefix_of []);
cons(e,s) is_prefix_of cons(e,s') = s is_prefix_of s';
e ≠ e' ⇒ ¬ (cons(e,s) is_prefix_of cons(e',s'));
```

Although written in SPECTRUM the above specification reflects the constructive character of the OBSCURE specification. The axioms of the function length (given in Section 3.2), for instance, constitute a primitive recursive definition and are therefore "executable". A similar remark holds for a function such as .∈. given a few lines above, being understood that the implication .⇒. corresponds to the if−then−else construct of a programming language. For a more precise formulation of the notion of constructivity the reader is referred to the chapter "Constructive Specifications" in [12].

4.3 The Specification REGEXP

This specification is identical with the requirement specification of Section 3.3.

4.4 The Specification MATCH

The design specification is identical with the requirement specification of Section 3.4 except for the axioms

```
(r1 ∘ r2) matches s = ...
```

and

```
(** r1)  matches s = ...
```

Moreover, it introduces the function

```
.is_match_of. : (Regexp × Regexp) × (String ×String) → Bool
```

instead of .is_prefix_match_of. of the requirement specification.

The axioms for .∘., ** and the new function are:

```
(r1 ∘ r2) matches s = (r1,r2) is_match_of ("",s);
(** r1) matches "";
(** r1) matches cons(c,s) = (r1,**r1) is_match_of (mklist(c),s);

(r1,r2) is_match_of (s1,"") = (r1 matches s1) ∧ (r2 matches "");
(r1,r2) is_match_of (s1,cons(c,s)) =
          (  (r1 matches s1) ∧ (r2 matches cons(c,s))   )
          ∨  ((r1,r2) is_match_of (s1++mklist(c),s));
```

4.5 The Specification SCAN

The specification SCAN is now:

```
SCAN = {

 enriches EXT_LISTS + MATCH;

 .list_matches. : List Regexp × String → Bool × Regexp;
 longest_prefix_match :
     String × List Regexp → Bool × String × Regexp;
 .list_matches., longest_prefix_match strict total;

 axioms ∀s,s':String, r:Regexp, rs:List Regexp, b:Bool in
   [] list_matches s = (false,∅);
   r matches s ⇒ cons(r,rs) list_matches s = (true,r);
   ¬(r matches s) ⇒ cons(r,rs) list_matches s = rs list_matches s;

   rs list_matches s = (true,r) ⇒
```

```
      longest_prefix_match(s,rs) = (true,¨¨,r);
  rs list_matches s = (false,r) ∧ length(s) ≤ 1 ⇒
      longest_prefix_match(s,rs) = (false,s,∅);
  rs list_matches s = (false,r) ∧ length(s) > 1
      ∧ longest_prefix_match(allbutlast(s),rs) = (b,s',r') ⇒
          longest_prefix_match(s,rs) = (b,s' ++ mklist(last s),r');
endaxioms;

scan  : String × List Regexp → List Regexp × String;
scan strict total;

axioms ∀s,s',s'':String, r:Regexp, rs,rs':List Regexp, c:Char in
  s = ¨¨ ⇒ scan(s,rs) = ([],¨¨);

  s ≠ ¨¨ ∧ longest_prefix_match(s,rs) = (true,s',r)
           ∧ scan(s',rs) = (rs',s'')
           ⇒ scan(s,rs) =(mklist(r) ++ rs',s'');

  s ≠ ¨¨ ∧ longest_prefix_match(s,rs) = (false,s',r)
           ⇒ scan(s,rs) = ([],s);
endaxioms;
}
```

Intuitively, the design decisions (ii) and (iii) of Section 3.5 are implemented by `longest_prefix_match` and `.list_matches.` respectively. The design decision (v) is implemented by the second axiom for `longest_prefix_match` in the module `SCAN` (lines 4, 5 in `axioms... endaxioms`): the condition `length(s)`$\leq$`1` interrupts the recursion when the string argument consists of a single character. Actually, the axioms of `longest_prefix_match` induce a search through the set of all non-empty prefixes of the string `s`. This search starts with the longest prefix, i.e. the string `s` itself and terminates at the latest with the prefix of length 1.

When compared with the requirement specification of Section 3.5 the design specification introduces an additional function, viz. `.list_matches.`. Moreover, the function `longest_prefix_match` is a genuine function while `.is_longest_prefix_match_of.` is a predicate.

Finally, it is interesting to note the following subtle difference between the two functions `.is_longest_prefix_match_of.` of the requirement specification and `longest_prefix_match` of the design specification: the latter accepts an empty string as the longest prefix only if the string considered is itself the empty string. More precisely, let $[r_1, \ldots, r_n]$ a list of regular expressions, such that (at least) one of these regular expressions matches the empty string ¨¨. Let r_{i_0} be the *first* regular expression matching ¨¨. Formally

$$[r_1, \ldots, r_n] \texttt{ list\_matches } ¨¨ \;=\; (\texttt{true}, r_{i_0})$$

must hold. Now let s be a string, such that there is *no* non–empty prefix s' of s which is matched by one of the r_i, i.e. for *each* non–empty prefix s' of s

$$[r_1, \ldots, r_n] \texttt{ list\_matches } s' \;=\; (\texttt{false}, \emptyset)$$

must hold. Then the following equations are valid:

- $(\ddot{}\,\ddot{}, r_{i_0})$ `is_longest_prefix_match_of` $(s, [r_1, \ldots, r_n]) \;=\; \texttt{true}$,
- `longest_prefix_match`$(\ddot{}\,\ddot{}, [r_1, \ldots, r_n]) \;=\; (\texttt{true}, \ddot{}\,\ddot{}, r_{i_0})$
- ... and (!): `longest_prefix_match`$(s, [r_1, \ldots, r_n]) \;=\; (\texttt{false}, s, \emptyset)$, if $s \neq \ddot{}\,\ddot{}$.

By the way, this difference is not intentional but is a result from arbitrarily made decisions in the implementation of design decision (v): in the requirement specification it is realized by the axioms for the function `scan`, in the design specification by the axioms for the function `longest_prefix_match`.

5 Verification of Coincidence of Both Specifications

5.1 Proof Obligations

The goal was to prove that the requirement specification of Section 3 and the design specification of Section 4 are "mutually consistent". To this end the following two properties were established. First it was shown that the design specification is consistent in that it terminates (step (iii) of Section 1.2). This property is called the property of *termination*. Second it was shown that any model of the design specification is a model of the requirement specification. This property of *refinement* establishes step (v) of Section 1.2.

Both properties were translated into sets of formulas, called *proof obligations*. Proving a set of proof obligations is necessary and sufficient to establish the corresponding property. The translation is based on the theory of modular systems described in [16] and [17]. It is performed automatically by the KIV system, when the two specifications are presented to it in the form of a modular system. The proof obligations for termination and refinement are now discussed successively.

To prove the property of termination, recursive functions are generated from the equations of the design specification. It is then shown that these define total functions, i.e. that any computation of a function value terminates. This generation is performed automatically in the same manner as in the Obscure system for rapid prototyping (step (iv) of Section 1.2). The termination property guarantees that the enrichments from `LIST` to `EXT_LISTS`, from `EXT_LISTS` to `MATCH` and from `MATCH` to `SCAN` are hierarchy persistent, i.e. that each model of `LIST` (resp. `EXT_LISTS`, `MATCH`) may be extended to a model of `EXT_LISTS` (resp. `MATCH`, `SCAN`); in fact, the semantics of the additional operations is then simply the semantics of the algorithms. The basic data types of lists, natural numbers, regular expressions and tuples are consistent by construction as they are all defined as initial data types with the `data`-construct of Spectrum.

To prove the property of refinement it is sufficient to prove that the axioms of the requirement specification are logical consequences of those of the design specification. In other words, if *AxReq* and *AxDes* denote the set of the axioms of the requirement specification and of the design specification respectively, one has to prove that $AxDes \models ax$ for each axiom ax of *AxReq*. These proof obligations may be divided into three groups corresponding to the specifications `EXT_LIST`, `MATCH` and `SCAN`. The obligations for `MATCH` and `SCAN` are as follows. *MatchAx* and *ScanAx* stand for the axioms of `MATCH` and `SCAN` respectively:

$$MatchAx \vdash \neg(\emptyset \text{ matches } s) \tag{1}$$

$$MatchAx \vdash \varepsilon \text{ matches } s = (s = \text{¨¨}) \tag{2}$$

$$MatchAx \vdash \text{mkreg(c) matches } s = (\text{mklist } c = s) \tag{3}$$

$$MatchAx \vdash r1 \circ r2 \text{ matches } s = \exists s1,s2.\ s = s1 +\!+ s2 \wedge r1 \text{ matches } s1 \wedge r2 \text{ matches } s2 \tag{4}$$

$$MatchAx \vdash r1 \mid\mid r2 \text{ matches } s = r1 \text{ matches } s \vee r2 \text{ matches } s \tag{5}$$

$$MatchAx \vdash (** \text{ r}) \text{ matches } s = (\ s = \text{¨¨} \vee \exists ss.\ s = \text{flatten ss} \wedge \forall s'.\ s' \in ss \Rightarrow r1 \text{ matches } s'\ ) \tag{6}$$

$$MatchAx \vdash (s,r1) \text{ is\_prefix\_match\_of } s' \Leftrightarrow r1 \text{ matches } s \wedge s \text{ is\_prefix\_of } s' \tag{7}$$

$$\begin{aligned} &ScanAx \vdash \\ &(s',t) \text{ is\_longest\_prefix\_match\_of } (s,rs) \Leftrightarrow \\ &t \in rs \wedge (s',t) \text{ is\_prefix\_match\_of } s \wedge \\ &\forall s1,t1.\ (s1,t1) \text{ is\_prefix\_match\_of } s \wedge t1 \in rs \wedge \\ &\quad (s1,t1) \neq (s',t) \Rightarrow \\ &\quad \text{length } s1 \ \$<\$ \text{ length } s' \vee \\ &\quad (\text{length } s1 = \text{length } s' \wedge \text{precedes\_in } (t,t1,rs)) \end{aligned} \tag{8}$$

$$\begin{aligned} &ScanAx \vdash \\ &\text{scan(s,rs)} = \text{mkres(ts,s')} \Rightarrow \\ &\quad s' \text{ is\_postfix\_of } s \wedge \\ &\quad (\forall r.\ r \in ts \Rightarrow r \in rs) \wedge \\ &\quad (ts = [] \Rightarrow\ \ s = s' \wedge \\ &\qquad \forall s'',r.\ \ r \in rs \wedge s'' \neq \text{¨¨} \Rightarrow \\ &\qquad\quad \neg\ (s'',r) \text{ is\_prefix\_match\_of } s) \wedge \\ &\quad (ts\ \ \neq\ \ [] \Rightarrow \\ &\quad \exists s1,s2.\ s = s1 +\!+ s2 \wedge \\ &\qquad \text{scan(s2,rs)} = \text{mkres(rest(ts),s')} \wedge \\ &\qquad (s1,\text{first(s)}) \text{ is\_longest\_prefix\_match\_of } (s,rs)) \end{aligned} \tag{9}$$

Since there are no definitions of .is_longest_prefix_match_of. and the function .is_prefix_match_of. in the design specification the following constructive specification is added to the design specification MATCH

```
(s,r1) is_prefix_match_of s' ⇔
   r1 matches s ∧ s is_prefix_of s'
```
(10)

and the following constructive specification to the design specification SCAN:

```
longest_prefix_match(s0, rs) = (true, s1, r) ⇒
       ( (s, r) is_longest_prefix_match_of (s0, rs) ⇔
       s0 = s ++ s1 )
```
(11)

```
longest_prefix_match(s0, rs) = (false, s1, r0) ⇒
       ( (s, r) is_longest_prefix_match_of (s0, rs) ⇔
       list_matches(rs,¨¨) = (true, r) )
```
(12)

5.2 The Verification

The verification proceeds in two steps. First, the KIV System automatically generates the proof obligations discussed in Section 5.1. These proof obligations are then tackled by a proof strategy which is based on the paradigm of tactical theorem proving and constitutes the kernel of the KIV System.

An overview of the results of the verification may be found in Table 7. The first line of the table contains the number of automatically generated *proof obligations* that ensure the properties of termination and refinement. The second line indicates the number of *lemmas* that were "invented" (and proved) by the proof engineer. The proofs required a number of "high level" *proof steps* displayed on the third line. Such high level proof steps include the application of an induction hypothesis, the insertion of a lemma, the unfolding of a function definition, etc. The proof engineer had to interact several times with the KIV system by selecting the proof rule to be applied next. The number of these *interactions* is shown on the fourth line. All other proof steps were performed automatically thanks to heuristics, yielding a degree of *automatization* indicated in the last line.

Actually, the table does not reflect the complete proof effort. For rewriting and simplification the system used 26 properties of lists and 11 properties of regular expressions. The proofs of these 37 properties required 415 additional proof steps and 88 additional interactions. [4]

The figures in Table 7 refer to the final versions of the requirement and design specification. Originally these specifications contained several adequacy errors

[4] It is interesting to note that the verification of SCAN does not require any properties of *match*: in this sense the scanner is generic.

EXT_LISTS		**MATCH**		**SCAN**	
Proof obligations	13	Proof obligations	9	Proof obligations	4
Lemmas	0	Lemmas	5	Lemmas	13
Proof steps	223	Proof steps	360	Proof steps	470
Interactions	28	Interactions	90	Interactions	63
Automatization	87.4 %	Automatization	75 %	Automatization	86.5 %

Fig. 7. Overview over the verification

that were discovered in the course of the verification. The reuse of proofs [18] allowed to minimize the additional proof effort necessary to correct these errors. Altogether the verification could be completed within 4 days.

5.3 Detecting Adequacy Errors

The verification led to the discovery of four adequacy errors.[5] The goal of this section is to indicate the nature of these errors and to comment on the effort of the proof engineer to detect them.

The first error occurred in both the requirement specification and the design specification. Instead of the axiom:

$$\texttt{(** r1) matches s = (s = ¨¨ } \vee \exists \texttt{ss. s = flatten ss} \wedge \texttt{ } \forall \texttt{s'. s`} \in \texttt{ ss} \Rightarrow \texttt{ r1 matches s')} \tag{13}$$

of Section 3.4 the requirement specification contained the much simpler axiom:

$$\texttt{(** r1) matches s = (s=¨¨ } \vee \texttt{ r1}\circ\texttt{(**r1) matches s)} \tag{14}$$

This axiom is not adequate because it allows a model in which, for example, the regular expression `(**(**r))` matches any string. The design specification contained the same erroneous axiom instead of the two axioms of Section 4.4. This adequacy error was found during the attempt to prove the termination property for the function `.matches.` with the KIV-System. The attempt yielded an unprovable goal and a proof analysis lead to the counterexample:

$$\texttt{(** (** mkreg(c))) matches mklist(d)} \tag{15}$$

where `c`, `d` stand for two arbitrary, different characters. This input leads to a nonterminating computation.

[5] To be specific the fourth error was introduced by the attempt to correct the third one.

The second error was located in the requirement specification. It became apparent, when during the proof of (8) for .is_longest_prefix_match_of. the following subgoal appeared:

$$\begin{array}{l}\texttt{t matches "" } \wedge \texttt{ t}\in\texttt{rs } \wedge \texttt{ t1 matches "" } \wedge \texttt{ t1}\notin\texttt{rs } \wedge \texttt{ t1}\neq\texttt{t} \\ \quad \Rightarrow \texttt{precedes\_in(t,t1,rs)}\end{array} \tag{16}$$

The precondition of (16) is satisfiable since it is always possible to find a regular expression t1 different from t, which matches the empty string "" and which is not member of rs. However, the goal is not provable, since t can precede t1 in rs only if both regular expressions are members of rs. The problem is that we have to deal with regular expressions we are not interested in, i.e. that are not member of the considered list of regular expressions *rs*. An inspection of the axiomatization of .is_longest_prefix_match_of. revealed that in the first version the additional condition t1 $\in$ rs was missing:

$$\begin{array}{l}\texttt{(s',t) is\_longest\_prefix\_match\_of (s,rs)} \Leftrightarrow \\ \qquad \texttt{t} \in \texttt{rs} \wedge \texttt{(s',t) is\_prefix\_match\_of s} \wedge \\ \qquad \forall\texttt{s1,t1. (s1,t1) is\_prefix\_match\_of s} \wedge \\ \qquad\qquad \texttt{(s1,t1)} \neq \texttt{(s',t)} \Rightarrow \\ \qquad\qquad \texttt{length s1} < \texttt{length s'} \vee \\ \qquad\qquad \texttt{(length s1} = \texttt{length s'} \wedge \\ \qquad\qquad\quad \texttt{precedes\_in(t,t1,rs))}\end{array} \tag{17}$$

This is an adequacy error which does not lead to an inconsistent specification but to an irritating counter-intuitive definition: the value of the function .is_longest_prefix_match_of. is false for all arguments. The error was detected during the attempt to prove the property of refinement for the specification SCAN after only 29 proof steps.

The third adequacy error was located in the design specification. Originally, the two last axioms defining the function longest_prefix_match in Section 4.5 were

$$\begin{array}{l}\texttt{rs list\_matches s} = \texttt{(false,r)} \wedge \texttt{length(s)} = \texttt{0} \Rightarrow \\ \quad \texttt{longest\_prefix\_match(s,rs)} = \texttt{(false,s,}\emptyset\texttt{)}\end{array} \tag{18}$$

$$\begin{array}{l}\texttt{rs list\_matches s} = \texttt{(false,r)} \wedge \texttt{length(s)} \neq \texttt{0} \\ \wedge\ \texttt{longest\_prefix\_match(allbutlast s,rs)} = \texttt{(b,s',r')} \Rightarrow \\ \quad\ \texttt{longest\_prefix\_match(s,rs)} = \\ \quad\ \texttt{(b,s'} \mathbin{+\!\!+} \texttt{mklist(last s),r')}\end{array} \tag{19}$$

In other words, the axioms contained the conditions length(s) = 0 and length(s) > 0 instead of length(s) $\leq$ 1 and length(s) > 1 respectively. In the original version a regular expression matching the empty string is considered a successful match, while in the new version it is not. The error was

found during the attempt to prove the termination of the function `scan`. In fact, the computation of a value such as

$$\texttt{scan(cons(a,¨¨),[}\varepsilon\texttt{])} \tag{20}$$

failed to terminate, because

$$\begin{array}{l}\texttt{longest\_prefix\_match(cons(a,¨¨),[}\varepsilon\texttt{])} = \\ \texttt{(true,cons(a,¨¨),}\varepsilon\texttt{)}\end{array} \tag{21}$$

i.e. because the argument in the recursive call did not "decrease". The correction adopted first consists in the modification of the `scan`-function: if the unfinished rest does not decrease in length, the scanning is terminated.

This means that the axiom

$$\begin{array}{ll} & \texttt{s} \neq \texttt{¨¨} \\ \wedge & \texttt{longest\_prefix\_match(s,rs) = (true,s',r)} \\ \wedge & \texttt{scan(s',rs) = (rs',s'')} \\ \Rightarrow & \texttt{scan(s,rs) = (cons(r,rs'),s'')} \end{array} \tag{22}$$

is replaced by the following two axioms:

$$\begin{array}{ll} & \texttt{s} \neq \texttt{¨¨} \\ \wedge & \texttt{longest\_prefix\_match(s,rs) = (true,s',r)} \\ \wedge & \texttt{s} \neq \texttt{s'} \\ \wedge & \texttt{scan(s',rs) = (rs',s'')} \\ \Rightarrow & \texttt{scan(s,rs) = (cons(r,rs'),s'')} \end{array} \tag{23}$$

$$\begin{array}{ll} & \texttt{s} \neq \texttt{¨¨} \\ \wedge & \texttt{longest\_prefix\_match(s,rs) = (true,s,r)} \\ \Rightarrow & \texttt{scan(s,rs) = (mklist(r),s)} \end{array} \tag{24}$$

However, with this version the proof of (9) for `scan` got stuck with the following subgoal:

$$\begin{array}{ll} & \texttt{longest\_prefix\_match(s',rs) = (true,s',r2)} \\ \wedge & \texttt{r2} \in \texttt{rs} \\ \wedge & \texttt{s'} \neq \texttt{¨¨} \\ \wedge & \texttt{r2 matches ¨¨} \\ \Rightarrow & \exists\texttt{s1,s2.}\ \ \texttt{s' = s1 ++ s2} \\ & \qquad \wedge\ \texttt{scan(s2,rs) = mkres([],s')} \\ & \qquad \wedge\ \texttt{(s1,r2) is\_longest\_prefix\_match\_of (s',rs)} \end{array} \tag{25}$$

The only possible instantiation for `s2` is `s'`, since `s'` is the unfinished rest of the scanning of `s2` and cannot be *longer* than `s2`. This requires `s1` to be instantiated by `¨¨` (`s' = s1 ++ s'` must hold). However, `scan(s',rs)` is equal to `mkres(cons(r2,[]),s')`, not `mkres([],s')`; hence this goal is not

provable. An analysis of the proof tree showed that the correction (Axioms 23 and 24) was not compatible with the requirement specification.

Originally, both specifications were drawn up on the basis of the design decisions (*i*) to (*iv*) of Section 3.5. In the absence of the design decision (*v*) (Section 3.5) the match of the empty string was treated differently in the requirement and in the design specification. This discrepancy was detected by the KIV-System during the proof of the refinement property for the specification `SCAN` after 110 proof steps. However, no work was lost because all proof steps could be reused automatically for the proof of the final version of the specifications, even though two functions and one lemma were modified. By the way, it was the detection of this error that led to the introduction of the design decision (*v*).

6 Conclusions

This paper has reported on a method which is intended to improve the quality of functional specifications in terms of precision and unambiguity. According to common experience from software practice this issue is of particular importance since errors made during the early steps of a project are the most expensive ones.

The quality of a specification may be judged by its logical consistency and by its adequacy. The basic difficulty in improving this quality is that there is no formally definable notion of the adequacy of a specification. The proposed method, called method of complementary specification, addresses this topic by introducing redundancy into the process of specification development. In the case study treated three groups of people were involved looking at the application problem from different viewpoints. The first group drew up a high-level requirement specification using a powerful logic language. The second group produced a design specification in the form of a very abstract model implementation for the same problem. While using the results of the first group it nevertheless produced a substantially different specification due to an algorithmic rather than an axiomatic "way of thinking". The third group, finally, established a formal relation between these two specifications. It thus improved the confidence in the adequacy of the specifications and — as a side effect — showed the consistency of the high-level specification.

The experiment described has shown that the method of complementary specification is a viable way to assess the quality of a formal specification. Although both specifications were carefully designed by specialists in formal methods, significant adequacy errors remained and were detected by applying the method — as described in Section 5.3. Moreover, the experiment has enabled the effective usage of software tools — such as OBSCURE and KIV — for the analysis of the specifications.

The twofold specification effort required by this method may seem a significant drawback. However, the following arguments show that it may make sense to have both specifications available in a project. The high-level requirement specification is more compact and better suited for the documentation of the problem. The executable design specification together with a symbolic interpreter is an ideal tool to decide cases in which there remains a doubt on the precise implications of the specification. None of the two specifications is satisfactory by itself: the high-level specification lacks good methods for checking its consistency, and the executable specification is too complex to decide upon its adequacy.

The method is particularly well-suited for problems of significant algorithmic complexity, the problem domain of which is well-defined. Of course, the method makes sense only if the quality requirements are high.

As a summary, for a project with a very high quality demand the method of complementary specification presented here seems to be a promising way to establish a sound starting point for the development. It is expected that the additional effort spent during the analysis of the specification will pay back by savings in the later development phases.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullmann. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] A. Ayari, S. Friedrich, R. Heckler, and J. Loeckx. Das Fallbeispiel LEX. Technical Report WP92/39, Uni Saarbrücken, 1992.

[3] 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.

[4] 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.

[5] Ch. Choppy. Is my Specification Correct? A Study with PLUSS Specifications. Technical Report 817, Univ. Paris-Sud, 1993.

[6] M. Heisel, W. Reif, and W. Stephan. Tactical Theorem Proving in Program Verification. In M. Stickel, editor, *International Conference on Automated Deduction.* Springer LNCS 449, 1990.

[7] M. Heisel, W. Reif, and W. Stephan. Formal Software Development in the KIV System. In R. McCartney and M. Lowry, editors, *Automating Software Design*. AAAI press, 1991.

[8] R. Hettler. A Requirement Specification for a Lexical Analyzer. Technical Report TUM-I9409, TU München, 1994.

[9] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, May 1992.

[10] Th. Lehmann and J. Loeckx. OBSCURE: A Specification Language for Abstract Data Types. *Acta Informatica*, 30(4):303–350, 1993.

[11] J. Loeckx. Algorithmic Specifications: A Constructive Specification Method for Abstract Data Types. In *ACM Trans. Progr. Lang. Syst. (TOPLAS) 9*, pages 646–685, 1987.

[12] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner Series in Applicable Theory in Computer Science. Wiley-Teubner. To appear.

[13] J. Loeckx and J. Zeyer. Experiences with the Specification Environment OBSCURE. Technical Report WP 93/48, Univ. Saarbrücken, 1993. Submitted for publication.

[14] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[15] W. Reif. The KIV-approach to software verification: state of affairs and perspectives. In *KORSO, Correct Software by Formal Methods*. Springer LNCS. To appear.

[16] W. Reif. Correctness of Generic Modules. In Nerode and Taitslin, editors, *Symposium on Logical Foundations of Computer Science*. Springer LNCS 620, 1992.

[17] W. Reif. Verification of Large Software Systems. In Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*. Springer LNCS 652, 1992.

[18] W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In Shyamasundar, editor, *Foundation of Software Technology and Theoretical Computer Science*, pages 284–293. Springer LNCS 761, 1993.

[19] D. T. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Coll. on Foundations of Computation Theory*, pages 413–427. Springer LNCS 158, 1983.