Software Engineering Journal, September 1992

Specifications Are (Preferably) Executable

Norbert E. Fuchs

Department of Computer Science
University of Zurich
fuchs@ifi.unizh.ch

The validation of software specifications with respect to explicit and implicit user requirements is extremely difficult. To ease the validation task and to give users immediate feedback of the behavior of the future software it was suggested to make specifications executable.

However, Hayes and Jones [Hayes, Jones 89] argue that executable specifications should be avoided because executability can restrict the expressiveness of specification languages, and can adversely affect implementations.

In this paper I will argue for executable specifications by showing that non-executable formal specifications can be made executable on almost the same level of abstraction and without essentially changing their structure. No new algorithms have to be introduced to get executability. In many cases the combination of property-orientation and search results in specifications based on the generate-and-test approach. Furthermore, I will demonstrate that declarative specification languages allow to combine high expressiveness and executability.

1 Introduction

In more than one way specifications play a central role in software development. They define all required characteristics of the software to be implemented, and thus form the starting point of any software development process. Specifications are also an important means of communication between users and developers – they can even be contracts. As formalized expressions of the requirements, specifications stand at the borderline between informal and formal descriptions of an application area.

To precisely and concisely define the required characteristics, specifications must be written in formal and highly expressive languages [Wing 90]. For an immediate reflection of the consequences of the specifications and for an early validation, it has been suggested that specifications should furthermore be executable [Agresti 86].

Hayes and Jones [Hayes, Jones 89] however, argue that the demands for high expressiveness and executability exclude each other, and that executable specifications should be avoided. In addition, they state that executable specifications can negatively affect implementations. They support their arguments by a detailed and exhaustive discussion of specification problems which in their opinion cannot adequately be handled by executable specification languages.

For two reasons I dissent. First of all, I consider the lack of correctness of software the most serious problem of software development, and not – as Hayes and Jones – the possible lack of expressive power of specification languages. Since the correct behavior of the software is defined by the requirements, and since the specifications are the basis for software development, all means applicable should be available to validate the specifications with respect to explicit and implicit requirements. Executable specifications can be crucial for this because they allow – in addition to formal reasoning about the specification – immediate validation by execution, and they provide users and developers with the *touch-and-feel* experience necessary to validate non-functional behavior, e.g. user interfaces. Excluding executability from specification languages means therefore depriving oneself of a powerful method for validation.
Second – as I will show in this paper – high expressiveness and executability need not exclude each other if specifications are written in declarative languages.

In chapter 2 of this paper, I will present the characteristics of executable specifications and a summary of Hayes' and Jones' critique. The chapter also contains a brief digression on logic specification languages. In chapter 3, I will address the question of expressiveness of specification languages vs. their executability. I will refute Hayes' and Jones' argument against executable specifications by demonstrating that their example specifications can be made executable on almost the same level of abstraction, and without essentially altering their structure. All executable specifications remain property-oriented and declarative. Chapter 4 is devoted to a detailed discussion of other aspects of executable specifications, namely the transformation of non-executable into executable specifications, the efficiency of the derived executable specifications, the constructiveness of specifications, the limits of abstraction, the specification and validation of non-functional requirements, inferences from specifications, and the distinction between executable specifications and implementations.. Some of these aspects are also addressed by Hayes and Jones. Chapter 5 summarizes the main results.

In the remainder of this paper *Hayes and Jones* stands for [Hayes, Jones 89].

2 Software Specifications

2.1 Executable Software Specifications

A software specification should describe the required behavior of a future software system in problem-oriented terms, i.e. the specification should form a conceptual model of the system. The description should be as abstract as the requirements permit. This is only possible if the specification language is sufficiently powerful to express the required behavior adequately.

Traditionally, specifications have been written in natural language, but today more and more specifications are written in formal specification languages [Wing 90]. Compared with specifications in natural language, formal specifications have many advantages [Gehani 86]. Since a formal language has a well-defined syntax and a well-defined semantics, all details of a specification must be stated explicitly; thus missing, ambiguous, or inconsistent information can much easier be found. In addition, formal reasoning about the specification is possible, specifically verification and validation with respect to the requirements. In the context of this paper, I will only consider specifications in formal languages. Descriptions in natural language will be called requirements.

If the formal specification language is executable there is an additional advantage: an executable specification represents not only a conceptual, but also a behavioral model of the software system to be implemented [Fromherz 89]. The behavior of the system interacting with its environment can be demonstrated and observed before it actually exists in its final form.

This quality of executable specifications promises to remedy the most serious problem of software – its lack of correctness and reliability. In the traditional software development one of the main reasons for this problem is the great time lag between the specification of a system and its validation. Since the implementation is the first formal – and executable – version of the system, validation is only possible after design decisions were made. This means that validation has to deal with the enormous amount of detail of the implementation. Possibly, design decisions have to be redone. Executable specifications, however, allow early validation on an abstract level and in terms of the problem. This will increase the correctness and the reliability of the software, and reduce development costs and time.

Furthermore, executable specifications can serve as prototypes which allow to experiment with different requirements, or to use an evolutionary approach for software development. This is especially important since in many projects the requirements cannot initially be stated completely and precisely.

Their very executability makes executable specifications an optimal communication vehicle between users and developers in their discussion of the intended system behavior. Even persons uncomfortable with formality can experience the behavior generated by executing the specifications and check whether it conforms to their intentions [Fromherz 89].

Executable specifications can be embedded in various software development paradigms. Especially attractive are the combinations with prototyping, and with the operational and the transformational approaches [Agresti 86]. If we combine executable specifications with the transformational approach the executable specification will form the only relevant document for all phases of software development.

2.2 Hayes' and Jones' Critique of Executable Specifications

Hayes and Jones argue that executable specifications should be avoided. Their arguments against executable specifications can be summarized in the following statements.

1. All formalisms applicable should be used to specify the desired properties of a system. Executability inevitably limits the expressive power of a specification language and restricts the forms of specifications that can be used. Specifications should be phrased in terms of required properties of the system. They should not contain the algorithmic details necessary to make them directly executable.

2. Though executable specifications permit early validation with respect to the requirements by executing individual test cases, proving general properties about a specification is much more powerful.

3. Executable specifications can unnecessarily constrain the choice of possible implementations. Implementors can be tempted to follow the algorithmic structure of the specification although that may not be desirable. Executable specifications can produce particular results in cases where a more implicit specification may allow a number of different results.

4. It will be easier to verify that an implementation meets a more abstract specification, than to match an executable specification against an implementation for which possibly different data and program structures have been chosen.

In the main part of their paper, Hayes and Jones elaborate on their arguments against executable specifications by addressing in detail particular aspects and problems of specifications. In each case, they argue that executable specifications are not able, or not adequate, to represent the pertinent aspect, or to solve the pertinent problem. They illustrate their arguments by a number of example specifications.

2.3 Logic Specification Languages

Declarative languages, e.g. the functional language ML [Milner et al. 90] or the logic language Prolog [Sterling, Shapiro 86], state *what* is to be computed in a form that is largely independent of *how* the computation is performed. Declarative languages are based on sound mathematical foundations, have well-defined semantics, permit descriptions at a very high level of abstraction, and are referentially transparent. Thus declarative languages are especially suitable as specification languages.

Traditionally, logic has been used as a powerful, concise, and declarative language for software specifications. The restriction to Horn clause logic and the mechanization of proofs which lead to logic languages like Prolog makes these specifications executable [Kowalski 85]. Logic specification languages have enjoyed much interest. Accounts of research directions in logic specification languages can be found in [Goguen 86] and in [Levi 86]. Recent publications on logic specification languages are e.g. [Fromherz 91], [Ghezzi et al. 90], [Stadler 90], [Terwilliger, Campbell 89], [Terwilliger 90], and [Ural 90].

In the following, I will express example specifications in a logic specification language – for conciseness called LSL – which is based on the extended Horn clause syntax proposed by Lloyd and Topor [Lloyd, Topor 84]. LSL can be considered a subset of the logic programming language Gödel [Hill, Lloyd 91].

A specification consists of a finite number of LSL statements of the form

```
H <- B.
```

where the head H is an atomic formula and the body B is a (not necessarily closed) first-order formula containing the usual connectives and quantifiers. Free variables in H and in B are implicitly universally quantified in front of the statement. The body B can be absent.

An example specification is the subset predicate $\subseteq$ defined by the statement

$$X \subseteq Ys \text{ <- } \forall Z \, (Z \in X \text{ -> } Z \in Y).$$

LSL's computation rule is similar to the one adopted for Gödel. The leftmost literal that can safely be executed is selected for execution. Specifically, negated literals are only executed when they are ground. The execution of literals can be delayed until explicit conditions are fulfilled. This permits to specify coroutining and thus to increase efficiency.

The greater expressiveness of LSL facilitates the formulation of specifications in two ways. First, LSL statements are closer to statements in natural language than Horn clauses. This makes it easier to translate software requirements – which are typically expressed in natural language – into specifications written as LSL statements. Second, the readability of the specifications is increased because LSL statements allow to state facts more directly and more concisely.

In spite of their greater expressiveness, LSL statements remain within the Horn clause subset of predicate logic. In fact, Lloyd and Topor [Lloyd, Topor 84] demonstrated that statements can easily be transformed into equivalent Horn clauses provided that negation as failure is safe. The above LSL statement for the subset predicate can be transformed into the Horn clauses

$$X \subseteq Y \text{ <- not } p(X, Y).$$
$$p(X, Y) \text{ <- } Z \in X \wedge \text{not } Z \in Y.$$

The connective *not* stands for (safe) negation as failure.

3 Hayes' and Jones' Example Specifications

Hayes and Jones request that specifications should abstractly define properties of systems with all available means of expression, and should not restrict possible implementations. Their central argument is that these requests are jeopardized if specifications are executable.

I will refute Hayes' and Jones' argument by showing that their example specifications can directly be translated into executable form on almost the same level of abstraction, and without essentially altering their structure.

As will be seen, the translation of non-executable specifications into executable ones is made possible by reformulating them in a declarative specification language, in this case LSL, and by adding constructive elements. Since no new algorithms need to be introduced to achieve executability many of the executable specifications are based on search. The executable specifications remain property-oriented and have a declarative semantics.

3.1 Specifying in Terms of Available Functions

Hayes and Jones observe that very often specifications can be constructed as combinations of simpler properties or operations, especially if these are already available in the specification language. They state that the operators of predicate calculus can be used to build up the combinations.

Hayes and Jones further point out that to specify a problem concisely the specification language must provide an adequate level of functionality and the necessary expressive power. They state that simple problems can be specified in conventional programming languages, while for the adequate specification of complex problems one needs an expressive power which is usually not available in executable languages.

I will counter their arguments by showing that declarative languages, specifically LSL, provide the functionality and the expressiveness to adequately represent their example specifications in executable form.

Following Hayes and Jones and as usual in mathematics, I will assume some simple predicates to be available. I will not indicate in each case whether these predicates are predefined in LSL, or can be constructed from more elementary predicates.

Set Union and Intersection

Hayes and Jones specify the *union* and *intersection* of two sets, *S1* and *S2*, in a language that makes the functions $\cup$ and $\cap$ available.

$$\text{Union} := \text{S1} \cup \text{S2}$$

$$\text{Intersection} := \text{S1} \cap \text{S2}$$

In logic languages these two functions are not usually available, but they can easily be specified starting from their basic definitions. Functions are represented as predicates with an additional last argument which stands for the value of the function. Sets are represented as ordered lists without duplicates, the empty set as the empty list. The predicates *union* and *intersection* can then be specified by the following LSL statements.

```
union(S1, S2, Union) <-
    set(Element, (member(Element, S1) ∨ member(Element, S2)), Union).

intersection(S1, S2, Intersection) <-
    set(Element, (member(Element, S1) ∧ member(Element, S2)), Intersection).
```

The predicates *set* and *member* are assumed to be available. The predicate *set* is a set constructor. The atom

```
set(X, p(X), Xs)
```

constructs the set

$$\text{Xs} = \{ \text{X} \mid \text{p(X)} \}$$

(Note: the LSL predicate *set* returns the empty list for the empty set, while Prolog's standard predicate *setof* fails.)

The predicates *union* and *intersection* now being available we can specify the *union* and *intersection* of two sets as abstractly as Hayes and Jones.

Update of Text File

In the next example, Hayes and Jones specify the update of a text file. Although they state that their specification could eventually be made executable, I believe that it is instructive to redevelop the specification in a logic language.

A file consists of a sequence of lines represented as the list

```
File = [Line | Lines]
```

Lines to be deleted are modeled as a set of line numbers which is represented as a list

Deletes = [LineNumber | LineNumbers]

and a sequence of lines to be added is represented as a list of lines

Adds = [Line | Lines]

Lines can be added before the first line of the file, and after each line of the file. The list of lines *AddBefore* will be added before the first line. *AddsAfter* is a list of lists of lines. It contains for every line of the file a list of lines to be added after that line, i.e. line numbers are mapped to lists of lines. Many of these lists will be empty since no lines have to be added.

The updated file is defined by Hayes and Jones as

$$\text{UpdatedFile} = \text{AddBefore}\ \&$$
$$\&\&\ \{\ (\text{if } N \in \text{Deletes then [] else File}(N))\ \&\ \text{AddsAfter}(N)\ \mid\ N \in \text{domain(File)}\ \}$$

where *&* denotes the concatenation of two sequences and *&&* the concatenation of all the sequences in the sequence of sequences of lines. Two preconditions

$$\text{Deletes} \subseteq \text{domain(File)}$$

$$\text{domain(AddsAfter)} = \text{domain(File)}$$

state that lines to be deleted have to be in the original file, and that additions must go after every line of the original file.

In LSL, I specify the update operation by the predicate *update*. The predicate is defined as an implication of the conjunction of its pre- and postconditions [Fuchs 92], and thus completely reflects Hayes' and Jones' original non-executable definition.

```
update(File, Deletes, AddBefore, AddsAfter, UpdatedFile) <-
   domain(File, DomainOfFile) ∧                        % domain(File)
   subset(Deletes, DomainOfFile) ∧                     % Deletes ⊆ domain(File)
   domain(AddsAfter, DomainOfFile) ∧                   % domain(AddsAfter) = domain(File)
   delete_add(File, DomainOfFile, Deletes, AddsAfter, IntermediateFile) ∧
   concatenate(IntermediateFile, ConcatenatedIntermediateFile) ∧
   append(AddBefore, ConcatenatedIntermediateFile, UpdatedFile).
```

The operations *&* and *&&* are represented by the predicates *append* and *concatenate*, respectively. The predicate *delete_add*

```
delete_add([LineN | Lines], [N | Ns], Deletes, [AddN | Adds], [NewLinesN | NewLines]) <-
   (if member(N, Deletes) then NewLineN = [] else NewLineN = [LineN]) ∧
   append(NewLineN, AddN, NewLinesN) ∧
   delete_add(Lines, Ns, Deletes, Adds, NewLines).
delete_add([], _, _, _, []).
```

calculates the sequence of sequences of lines

$$\{\ (\text{if } N \in \text{Deletes then [] else File}(N))\ \&\ \text{AddsAfter}(N)\ \mid\ N \in \text{domain(File)}\ \}$$

by a recursive loop over the domain of the file.

The predicates *domain*, *subset*, *concatenate*, *append*, *member*, and the operator notation of *if-then-else* are assumed to be available.

The example update operation

update([f1, f2], [2], [a0], [[a1], []], UpdatedFile)

results in

UpdatedFile = [a0, f1, a1]

And now for something completely different ...

Sorting Sequences Without Duplicate Elements

Hayes and Jones specify the *sort* operation for sequences of natural numbers without duplicates using the simpler *permutation* and *ordered* properties. The *sort* operation transforms a sequence X satisfying the precondition

sequence_of_natural(X)

into a sequence Y which satisfies the postcondition

permutation(X, Y) $\wedge$ ordered(Y)

If we assume – like Hayes and Jones – that the predicates *sequence_of_natural*, *permutation* and *ordered* are available we can immediately specify the predicate *sort* in LSL as an implication of the conjunction of its pre- and postconditions

sort(X, Y) <- sequence_of_natural(X) $\wedge$ permutation(X, Y) $\wedge$ ordered(Y).

This well-known *slow sort* algorithm is in the form of a generate-and-test solution. The executable specification is as property-oriented and declarative as the non-executable one.

Greatest Common Divisor and Least Common Multiple

Hayes and Jones specify the greatest common divisor of two natural numbers by

gcd(I, J) = D <=> common_divisor(D, I, J) $\wedge$ $\neg$ ($\exists$ E $\in$ N (common_divisor(E, I, J) $\wedge$ E > D))

common_divisor(D, I, J) <=> divides(D, I) $\wedge$ divides(D, J)

and state that the structure of the specification does not lead directly to the structure of a program to calculate the greatest common divisor. They further state that to make the specification executable it would be necessary to complicate the specification, and to reason about the problem.

I will show that the specification can be made executable without changing its structure, without complicating it, and with only some reasoning about termination.

To transform this specification into an executable one I represent the function *gcd* by the predicate *gcd* defined by the LSL statement

gcd(I, J, D) <- D $\in$ N $\wedge$ common_divisor(D, I, J) $\wedge$ not (E $\in$ N $\wedge$ common_divisor(E, I, J) $\wedge$ E > D).

I introduced the precondition $D \in N$ that is implicit in the original specification, and replaced the logical negation $\neg$ by the negation as failure *not*. I also omitted the existential quantifier for E since all free variables in LSL statements are implicitly quantified.

To make the specification of *gcd* executable I represent the predicate ∈ by the executable predicate *limited_natural_number(Natural,Limit)* which recursively generates natural numbers *Natural* up to *Limit*. Since the largest integer divisor of an integer is the integer itself, the domains of the variables *D* and *E* are finite. The generator uses these obvious upper limits to enforce termination of the generate-and-test cycle. We get the executable specification

```
gcd(I, J, D) <-
    limited_natural_number(D, I) ∧
    common_divisor(D, I, J) ∧
    not (limited_natural_number(E, I) ∧ common_divisor(E, I, J) ∧ E > D).

common_divisor(D, I, J) <- divides(D, I) ∧ divides(D, J).
```

which directly reflects the original specification.

In the same way I translate the specification of the *least common multiple*

$$\mathrm{lcm}(I, J) = M \Leftrightarrow \mathrm{common\_multiple}(M, I, J) \land \neg\, (\exists L \in N\ (\mathrm{common\_multiple}(L, I, J) \land L < M))$$

$$\mathrm{common\_multiple}(M, I, J) \Leftrightarrow \mathrm{divides}(I, M) \land \mathrm{divides}(J, M)$$

into the LSL statements

```
lcm(I, J, M) <-
    limited_natural_number(M, I * J) ∧
    common_multiple(M, I, J) ∧
    not (limited_natural_number(L, I * J) ∧ common_multiple(L, I, J) ∧ L < M).

common_multiple(M, I, J) <- divides(I, M) ∧ divides(J, M).
```

In the definition of *lcm* I use the fact that the least common multiple of two integers is not larger than their product. As a consequence, the generated set of common multiples is finite.

The results show that non-executable specifications of the *greatest common divisor* and of the *least common multiple* can directly be translated into LSL. The resulting executable specifications are based on the generate-and-test technique and remain declarative.

During the translation I replaced the predicate ∈ by a recursive generator which generates natural numbers up to a given limit, i.e. I used a specific property of the problem to control the enumeration and to enforce termination. Hoare uses the same reasoning in his paper on formal methods [Hoare 87]. Hayes and Jones doubt the generality of this approach. They argue that it can be difficult to find such a property, and if found the property is possibly not as simple as in the example.

This can actually be the case. On the other hand, it is always possible to enforce the termination of an enumeration by an educated guess. A similar guessing is used in limited depth-first search. Iterative deepening shows how the guessing can even be automated [O'Keefe 90].

3.2 Specifying by Inverse

Sometimes it is impossible to specify an unknown function directly in terms of given functions. Hayes and Jones state that in this case the function can possibly be specified indirectly by constraining its inverse by available functions. They emphasize that inverse functions are not generally executable, or if executable, that the execution is usually very inefficient.

I will show that LSL allows specification by inverse. Contrary to Hayes' and Jones' supposition the inverse specifications are executable, but they are based on search and can indeed be rather inefficient.

Integer Square Root

Hayes and Jones define the unknown function *integer square root* by its known inverse *square*. The natural number *R* as largest integer square root of the natural number *I* is defined by the constraints

$$R^2 \leq I < (R+1)^2$$

I define the predicate *integer_square_root* again as an implication of the conjunction of its pre- and postconditions

integer_square_root(I, R) <- R ∈ N ∧ R**2 ≤ I ∧ I < (R+1)**2.

Next, I introduce a representation for the predicate ∈. Since termination is guaranteed by the constraints, I represent ∈ by the recursive generator *natural_number* which enumerates all natural numbers.

integer_square_root(I, R) <- natural_number(R) ∧ R**2 ≤ I ∧ I < (R+1)**2.

This executable specification defines *integer square root* declaratively by its properties, and generates solutions by a simple generate-and-test approach.

Parse Tree

Hayes and Jones observe that it can be convenient to express the task of constructing a parse tree by demanding that the concatenation of the terminal strings of the wanted tree yields the input string. They point out that – if this inverse specification is executable at all – it leads to enormously inefficient search.

Like Prolog, LSL provides definite clause grammars in the form of grammar rules, e.g.

```
sentence      -->  noun_phrase, verb_phrase.
noun_phrase   -->  determiner, noun.
verb_phrase   -->  verb, noun_phrase.
determiner    -->  [the].
noun          -->  [cat] | [dog].
verb          -->  [sees] | [bites].
```

The grammar rule

```
sentence      -->  noun_phrase, verb_phrase.
```

is understood as the LSL statement

sentence (S0, S) <- noun_phrase(S0, S1) ∧ verb_phrase(S1, S).

As a consequence a grammar given in the form of grammar rules is executable; it is already a recursive-descent parser for the language it defines. The grammar is reversible: it allows to parse given sentences and to generate non-deterministically all sentences that can be derived from the grammar.

By adding to the non-terminals arguments that represent the respective subtrees it is possible to construct the parse tree automatically during the parsing, e.g.

```
sentence(s(NP, VP))          -->  noun_phrase(NP), verb_phrase(VP).
noun_phrase(np(DET, N))      -->  determiner(DET), noun(N).
verb_phrase(vp(VP, NP))      -->  verb(VP), noun_phrase(NP).
…
```

Parse trees are then represented as compound terms, e.g. the parse tree of the sentence

the dog sees the cat

by the term

s(np(det(the), n(dog)), vp(v(sees), np(det(the), n(cat))))

Following Hayes' and Jones' proposal, we can construct the parse tree P0 of a given sentence S0 by requiring that the concatenation of the terminals of P0 equals S0. In the conjunction

sentence(P, S, []) ∧ S = S0

sentence(P, S, []) generates – by virtue of the reversibility of the grammar – all possible sentences *S* with their respective parse trees *P*, while *S = S0* tests whether *S* equals the given sentence *S0*. If the test succeeds *P* is the wanted parse tree *P0*.

As Hayes and Jones point out, this generate-and-test approach is enormously inefficient because many possible sentences with their respective parse trees are generated before the correct sentence and parse tree are found. Fortunately, it is not necessary to specify the construction of the parse tree indirectly since – as we just saw – it can be specified directly with no more search involved than required by recursive-descent parsing.

3.3 Non-Computable Clauses in Specifications

Hayes and Jones argue that a specification language should be expressive enough to specify in the same notation computable problems, non-computable problems (e.g. the halting problem), and problems that are not known to be computable (e.g. Fermat's last theorem). Especially interesting are specifications that contain components which are not computable by themselves. When these components are combined with constraints the whole can become computable. Hayes and Jones state that in this case the structure of the specification does not lead directly to the structure of the implementation, as components are not computable.

Using their example I will show that their non-executable specification leads directly to an executable one.

Hamming Numbers

Hamming numbers are those natural numbers whose only prime factors are 2, 3, and 5 [Dijkstra 76]. The problem is to generate the sequence of Hamming numbers in increasing order. Hayes and Jones specify the sequence of Hamming numbers by

ordered(Hamming) ∧ Hamming = $\{$ H ∈ N | P divides H -> P ∈ {2, 3, 5} $\}$

The sequence of Hamming numbers is infinite and cannot be computed, but – as Hayes and Jones point out – finite prefixes of the sequence are computable.

The predicate *hamming_number* is true if *H* is an individual Hamming number.

```
hamming_number(H) <-
        natural_number(H) ∧
        has_primefactors_2_3_5(H).
```

I represent the pre- and postconditions by the executable predicates *natural_number* and *has_primefactors_2_3_5*. Based on this definition, the predicate *hamming_numbers* calculates the ordered sequence *Hamming* of Hamming numbers represented as a list.

```
hamming_numbers(Hamming) <-
        set(H, (natural_number(H) ∧ has_primefactors_2_3_5(H)), Hamming).
```

The sequence *Hamming* being infinite, this predicate does not terminate. By replacing *natural_number* by *limited_natural_number*, I get the predicate *prefix_of_hamming_numbers(Limit,Hamming)* which calculates the finite sequence of Hamming numbers *Hamming* which are smaller than or equal to *Limit*.

```
prefix_of_hamming_numbers(Limit, Hamming) <-
        set(H, (limited_natural_number(H, Limit) ∧ has_primefactors_2_3_5(H)), Hamming).
```

The postcondition *ordered(Hamming)* is automatically fulfilled. For completeness it should nevertheless appear in the specification as a comment.

With the help of the predicate *prefix_of_hamming_numbers* I can for example calculate the Hamming numbers up to 20

2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20

The predicate *hamming_numbers* directly reflects the structure of the non-executable definition which is not changed besides representing a sequence by a list. Thus, my specification stays much closer to the original definition than Turner's specification [Turner 85] which introduces additional algorithmic details by generating and merging three streams of numbers. A recursive solution for the Hamming problem can be constructed by induction [Fuchs 92].

3.4 Non-Deterministic Operations

Non-deterministic operations can produce more than one result for a given input. Hayes and Jones emphasize that the semantics of a specification language needs to cover non-determinism, even if the final implementation is deterministic. They believe that executable specification languages do not have the required semantics, so that executable specifications can restrict the range of possible implementations by producing particular results in cases where a non-executable specification might allow a number of different results.

Logic languages are based on relations, and can thus be used to express non-deterministic operations.

Sorting Sequences With Duplicate Elements

As an example of a system that shows external non-determinism, Hayes and Jones generalize the sorting problem of section 3.1 by allowing duplicate elements, e.g. records with duplicate keys. Thus the sorting operation is no longer deterministic. Hayes and Jones demand that the specification respects this non-determinism and does not overly restrict the implementation (e.g. insertion sort will leave records with identical keys in the given order, while quicksort will not necessarily do this).

I represent records as *(key, value)* pairs. It is easy to generalize the LSL specification of the sorting operation (cf. section 3.1) so that it sorts sequences with duplicate elements. In fact, if we represent sequences as lists and define the predicate *ordered* as

```
ordered([]).
ordered([X]).
ordered([(Key1, Value1), (Key2, Value2)|Xs]) <-
    Key1 < Key2 ∧
```

ordered([(Key2, Value2) | Xs]).

the only change necessary to allow for records with duplicate keys is the replacement of the comparison operator < by ≤. Hayes and Jones introduce the same change in their non-executable specifications.

Sorting the sequence

[(2, b), (2, c), (1, a)]

gives the two possible results

[(1, a), (2, b), (2, c)]
[(1, a), (2, c), (2, b)]

Differential Files

Next, Hayes and Jones resume the file update problem of section 3.1. Given two files, they define a differential file as the sets of deletions and additions that transform the first file into the second. In general, the differential file is not unique. For example, if the original file contains two consecutive identical lines, and the updated file only one of these lines, either line could have been deleted.

Hayes and Jones point out that the specification of the differential file is an inversion of the specification of the update problem, and that considerable detail must be supplied to make the specification of the differential file executable. I will show that this need not be the case.

The LSL specification of the update problem is in the form of a relation of its arguments. Once some arguments are sufficiently instantiated, other arguments can be determined. Calling *update* with instantiated *File* and *UpdatedFile* will determine the differential file consisting of *Deletes*, *AddBefore*, and *AddsAfter*. Solutions are found non-deterministically by generate-and-test.

Using Hayes' and Jones' example of a file with two identical lines

update([f1, f1], Deletes, Before, After, [f1])

we get the expected two differential files

Deletes = [1]   Before = []   After = [[], []]
Deletes = [2]   Before = []   After = [[], []]

In fact, the degree of non-determinism is even larger, since there are three additional differential files which delete both lines, and subsequently add one of them in different places.

Internal Non-Determinism

Even if the external behavior of a system is defined to be deterministic, components can internally behave non-deterministically. A well-known example is a system of concurrent, cooperating processes. Hayes and Jones emphasize that specifications which define externally deterministic behavior by internal non-determinism are extremely useful since they leave great freedom for the implementation.

The specifications of the greatest common divisor and the least common multiple (section 3.1) demonstrated that logic languages, specifically LSL, can specify externally deterministic behavior with non-deterministic components. This applies especially to solutions that rely on the generate-and-test technique. Also other approaches – e.g. those based on the simple but powerful concept of non-deterministic finite automata – can generate deterministic behavior from a non-deterministic one.

Yet another form of internal non-determinism and external determinism arises when we interpret the literals *permutation(X, Y)* and *ordered(Y)* of the statement

sort(X, Y) <- sequence_of_natural(X) $\wedge$ permutation(X, Y) $\wedge$ ordered(Y).

as parallel processes that communicate via the shared variable *Y*. This interpretation forms the basis of concurrent logic languages, and of coroutining available in LSL, in Gödel, and in some Prolog implementations.

4 All Things Considered

After demonstrating that expressiveness and executability of logic specification languages do not exclude each other, I will now discuss various other aspects of executable specifications. Some of these aspects are also addressed by Hayes and Jones.

4.1 From Non-Executable To Executable Specifications

In the preceding chapter I demonstrated that non-executable specifications can be made executable by reformulating them in a logic specification language, and by adding a small number of *constructive elements*.

Specifically, the constructive elements were: representation of sets and sequences by lists, construction of sets and sequences by recursion, partial instantiation of variables, and representation of the predicate $\in$ by generators of elements of the respective domains. Generators are usually recursively defined, though for a finite domain its extension could be used. Some recursive generators need upper limits to ensure termination of the specification. Transformations that make non-executable specifications executable are also discussed by Partsch [Partsch 90].

Executable specifications generated in this way are direct translations of their non-executable counterparts. Since they are built from available powerful predicates they are property-oriented, declarative, and highly abstract, though the addition of constructive elements results in a level of abstraction slightly lower than that of non-executable specifications. Also the structure of the non-executable specifications is not essentially changed, since no new algorithms have to be introduced to achieve executability. In many cases the combination of property-orientation and backtracking results in specifications based on the generate-and-test technique.

This means that the transformation of non-executable into executable specifications is accompanied by a minimum of design decisions. But even these design decisions are revisable, they need not restrict possible implementations.

I want to emphasize that the translation of non-executable specifications into executable ones on a high level of abstraction is only possible because I use a declarative specification language, in this case the logic specification language LSL. Declarative languages are especially suitable for specifications because they provide the degree of functionality and the expressiveness to adequately formulate conceptual as well as behavioral models.

Logic languages like LSL are basically relational languages, and as shown in the preceding chapter can be used to specify non-deterministic operations – even if they have an infinite number of solutions – or to specify deterministic operations by constraining the non-determinism. Both non-determinism and determinism are available at any level of abstraction.

4.2 Efficiency of the Derived Executable Specifications

Many of the derived executable specifications are based on generate-and-test, i.e. on potentially inefficient search. Though the purpose of this paper is not to advocate generate-and-test solutions in

general, but to show that non-executable specifications can be transformed into equivalent executable ones, it is nevertheless interesting to show how generate-and-test solutions can be made more efficient.
For concreteness, I will use the specification of *sort* (cf. section 3.1)

sort(X, Y) <- sequence_of_natural(X) $\wedge$ permutation(X, Y) $\wedge$ ordered(Y).

as an example.

As specified, the generator *permutation* will generate complete permutations *Y* even if already the first few elements of *Y* are not ordered. This suggests the following coroutining which will reduce the runtime of *sort* considerably: let *permutation* generate the first two elements of *Y*, then execute *ordered*. If these elements are ordered let *permutation* generate the next element and continue testing, else let *permutation* generate another couple of elements. As a result of this interleaving of generator and tester, failure branches of the search tree are pruned much earlier, and the program becomes significantly more efficient than without coroutining.

While coroutining interleaves the generator and tester at run-time, program transformations based on unfold/fold rules [Gardner & Shepherdson 89] lead to a form of interleaving at compile-time. Unfolding *sort* on *permutation* and *ordered*, and folding the resulting conjunction *permutation(X,Y)* $\wedge$ *ordered(Y)* produces an equivalent, recursive program for *sort*. This program is considerably more efficient because a number of proof steps are performed before the actual execution, and constraints between arguments are already taken into account.

4.3 A Constructive Approach To Executable Specifications

Hayes and Jones specify the square root r_1 of a floating point number x by

$$x \geq 0 \wedge \mathrm{sqrt}(x) = r_1 \Longleftrightarrow \exists\, r \in R\ (r^2 = x \wedge |r_1 - r| < 0.01)$$

The symbol r represents a real number, r_1 its floating point approximation.

The specification postulates the existence of the square root, but is not constructive: there is no indication how the square root should be calculated. The specification cannot even be used to test a hypothetical solution.

Actually, the specification as given is incomplete. Hayes and Jones point out that to find an implementation for this specification programmers have to refer to their knowledge of the theory of real numbers. Thus the real numbers in the specification do not serve any direct functional purpose; they have to be understood as specification variables (cf. section 4.5) meaning 'use the theory of real numbers', and constraining possible solutions.

This implicit reference to a body of knowledge seems to pose no problem since we are dealing with the well-known and well-defined theory of real numbers. In my opinion, this approach cannot be generalized. In ill-defined situations – e.g. if we develop expert systems – the implicit reference to knowledge has turned out to be very problematic. Furthermore, the reference to the theory of real numbers is so much taken for granted that one tends to overlook that in other situations there may be no indication where and how to acquire the knowledge. As experience shows, these problems of knowledge acquisition become even worse when several people are involved since everyone may have different ideas about the undocumented information.

We can explicitly acknowledge that an application field is ill-defined, and use an evolutionary approach to develop a system [Stolze et al. 91]. We start by implementing only a part of the system, and let users work with this partial system. From the experience gained we derive further requirements, and augment the system until we are satisfied. But even in this approach the specifications of the initial system and of the additions must be clear and complete. Thus the problem of knowledge acquisition persists, though on a somewhat smaller scale.

As we have seen, formal specifications just as mathematical definitions can postulate the existence of an object without telling us how to construct it. I am convinced that a specification as an abstract definition of something that will have to be concretely realized must be constructive in the sense of *constructive mathematics*. Quine [Quine 70] defines as constructive a mathematics which is intolerant of methods affirming the existence of things of some sort without showing how to find them.

Constructive methods – especially constructive logics – are also proposed by other researchers ([Bry 89], [Bundy et al. 90], [Deville 90], [Flener 91], [Fribourg 90], [Lau, Prestwich 91]).

4.4 Degree of Abstraction of the Specification

Borrowing a saying of Einstein's, I maintain that specifications should be as abstract as possible, but not more abstract. I see three limitations to the degree of abstraction.

First, a specification as an adequate formalization of the requirements cannot be more abstract than the requirements themselves. If a specific algorithm is required, this algorithm must be specified. This argument applies as well to non-functional requirements constraining possible implementations. Some constraints can appear as comments in specifications, e.g. the requirement that a specific language should be used for the implementation. Other constraints, however, must be concretely specified, e.g. the requirement that the future software system has to adhere to the data structures of a given interface.

The second limitation to abstraction arises when we make formal specifications executable. Even if the degree of abstraction of the data structures and the algorithms stays the same, we have to add constructive elements to the executable specifications which are not present in the non-executable specifications.

User interfaces often account for a large percentage of the functionality of software systems. Their specification poses a third limitation to abstraction since they have to be specified in great detail. This limitation can be partially removed if – in addition to the usual predefined functions – powerful I/O functions are available to specify user interfaces abstractly. Fromherz [Fromherz 91] proposes an object-oriented specification framework that provides default I/O operations which can be used as defaults, tailored to the required functionality, or replaced by custom operations.

One distinguishes between property-oriented and model-oriented specifications [Wing 90]. A property-oriented specification defines the behavior of a system indirectly by a set of properties in the form of axioms that the system must satisfy. A model-oriented specification defines the behavior of a system directly by constructing a model of the system.

Often the level of abstraction determines whether a specification can be called property-oriented or model-oriented. If we specify the sorting algorithm by the two concurrent processes *permutation* and *ordered* as

```
sort(X, Y) <- permutation(X, Y), ordered(Y).
```

the specification can be called property-oriented. If we introduce explicit details of the process synchronization and of the data exchange, the specification could be called model-oriented. Since LSL allows to formulate both property-oriented and model-oriented specifications in the same language there does not seem to be a clear-cut boundary between them.

4.5 Specification of Non-Functional Requirements by Specification Variables

How does one specify non-functional requirements like efficiency, reliability, modularity, maintainability, ease-of-use, or constraints on resources? Many of these non-functional requirements define emerging properties of the implementation, others constrain possible implementations, and all can lead to demands on the software development process.

Traditionally, non-functional requirements are specified informally as comments, which are to be used by developers during the design and implementation phases. But there are other ways.

Hayes and Jones state that it is useful to introduce into the specification variables that do not stand for implementation variables. These *specification variables* do not play a part in the execution of the implementation, instead they can serve to formalize comments or directives for the development process, or they can be used to reason about the specification. In particular, specification variables allow to formalize non-functional requirements, and to validate them by formal proofs. However, it must be emphasized that some non-functional requirements defy formalization, e.g. ease-of-use of user interfaces. Their correct implementation can only be tested and must be validated by the user.

Hayes and Jones question the role of specification variables for executable specifications. Must executable specifications already fulfill constraints expressed by specification variables? How are specification and implementation variables related when specifications are executable?

In my opinion, specification variables can play an important role in executable specifications. I suggest that specification variables partake in the execution of the specification in the same way as other variables. The specification may or may not fulfill the specified constraints. In either case, valuable information will be gained that can help to validate and to verify the specification. This information is especially valuable if the executable specification is used for explorative prototyping. It can also result in guidelines for the development of the implementation.

4.6 Validation of the Specification

Correctness of a software system means correctness with respect to the requirements, i.e. with respect to explicit and implicit user intentions and needs. This implies that users must be involved in the validation of the system, since only they know what they want. Users prefer to think in problem-oriented terms, they are usually not interested in implementation details. This suggests that the conceptual level provided by specifications is the appropriate level for the user involvement, and that validation should preferably take place in the specification phase.

Validation means checking the correspondence between informal requirements and formal specifications. Principally, there can be no formal method to check this correspondence [Hoare 87]. This means, that we have to rely on other means to convince ourselves of the correctness of the specifications with respect to the requirements, viz. inspection, reasoning and execution.

Formal, non-executable specifications can be validated by reasoning. Unfortunately, this does not apply to user interfaces where the direct experience of operations and of timing is the major criterion for acceptability. Most specification examples, e.g. the ubiquitous greatest common divisor, or the standard stack example, emphasize the algorithmic aspect of specifications, i.e. the functional behavior. Considering that a growing percentage of the code of current programs pertains to input and output, i.e. nonfunctional behavior, the emphasis on functional behavior seems most unfortunate. There are other – still relatively small – specification examples, like the library data base [Kemmerer et al. 87], that could provide a more realistic point of view.

Since validation cannot go beyond the conviction that the specifications are correct, I believe that it is extremely important to have available all means that can be used for validation, i.e. in addition to inspection and reasoning, also the execution of specifications. Executable specifications can contribute enormously to the validation because they give users the necessary *touch-and-feel* experience. Even if an executable specification is so inefficient that it cannot be executed for all values, the information gained from 'partial execution' remains valuable.

Executable specifications result in a greater involvement of the users. Users can participate in the formulation of the specifications and in the immediate validation. The involvement of the users and the immediate reflection of the consequences of the specifications back to the user are of utmost importance because they contribute to the correctness of the software, to the reduction of the costs, and to the adherence to the schedules.

4.7 Inferences from Specifications

Hayes and Jones emphasize the relevance of being able to reason about a specification. Reasoning helps to validate the specification with respect to user requirements by inferring that the specification has the required properties. Reasoning also helps to verify that an implementation meets the specification, or to derive the implementation from the specification via transformations or refinement steps. Hayes and Jones maintain that in both cases the reasoning does not usually benefit from the fact that the specification is executable.

I concur with Hayes' and Jones' statement that traditional validation of software by individual test cases has inherent limitations. There is no way to predict the general behavior of a software system from the observed behavior of a finite number of test cases, and for non-deterministic systems test results are not necessarily reproducible.

However, I want to emphasize that testing of executable specifications – though basically subject to the same limitations – is in three ways different from testing implementations.

First, test results are of much greater relevance because they are available at the very beginning of software development; costly mistakes can be avoided.

Second, testing executable specifications is more efficient since it is done in terms of the application domain, and on a much higher, much more abstract level.

Third, if a logic specification language is used, executing a test case is actually a proof; testing and inference become identical. Some proofs have the same limited meaning as testing, while other proofs are more meaningful, e.g. if a predicate has a finite number of solutions, we can enumerate them, and prove that there are no other solutions.

Other forms of reasoning are available to prove properties of the specification and to verify it, e.g. to show its consistency. Meta-interpreters can derive information that is not directly related to specified properties, or can provide alternative proof methods. Declarative debugging [Lloyd 87] can locate the exact cause of errors. These forms of reasoning are only available because the specification is executable.

But reasoning as a means for validation has also its limitations. Usually, non-functional behavior cannot be validated by inference, it can only be tested. This applies especially to user interfaces which often form a large part of the requirements.

I agree with Hayes and Jones that the executability of a specification contributes little to the derivation of an implementation, or to the verification of the implementation with respect to the specification. It is usually not feasible to check that an executable specification and an implementation produce exactly the same results. (Cf. section 4.8 for a discussion of the transformational approach which constructively solves the verification problem.)

4.8 Executable Specifications vs. Implementations

Logic languages can be used to formulate executable specifications and implementations. Both uses must be clearly distinguished.
Primarily, there is a distinction by intent. Executable specifications form the basis for implementations, they are not themselves implementations. Problem-oriented and implementation-oriented phases of the software development should be clearly separated.

Executable specifications and implementations have different commitments to specific algorithms, specific data structures, and efficiency. If we remain on the abstraction level of non-executable specifications and do not introduce additional algorithms, executable specifications are often based on generate-and-test, i.e. on search ([Fuchs 92], [Levi 86]). As a consequence the execution of an executable specification is usually much less efficient than the execution of an implementation, though it may appear to be similar. Implementations, on the other hand, should efficiently generate the specified behavior in the targeted environment using all available algorithms and data structures.

This view is shared by Zave and Yeh ([Zave, Yeh 81], [Zave 82]) who state that the borderline between executable specifications and implementations is resource management. Specifications should only specify the functional and performance properties of the proposed system. Implementations, however, have to meet performance goals in the intended environment by an optimal use of resources.

The separation into problem-oriented and implementation-oriented phases is also reflected in another aspect. Specifications should be expressed in terms of the problem domain, and they should be abstract, concise, and extremely readable. Implementations, on the other hand, are usually much less readable because of their amount of detail and their optimized structure.

The transformational approach [Agresti 86] is an especially attractive way to generate an implementation. With the help of transformations we can gradually introduce refinements and details into an executable specification, and in this way transform it step by step into the implementation. We may even start with an incomplete specification. If the transformation steps preserve the meaning of the specification the verification of the implementation becomes unnecessary, i.e. the approach is constructive. For declarative languages the unfold/fold transformations ([Burstall, Darlington 77], [Tamaki, Sato 84], [Gardner, Shepherdson 89]) can be used. These source-to-source transformations can change data structures and algorithms, and can generate implementations that are orders of magnitude more efficient than the specification. If we succeed in automating the transformations to a large extent [Fuchs, Fromherz 91] the executable specifications will form the only relevant document for software development.

In this context it is worthwhile to briefly discuss prototypes. Many people consider executable specifications as prototypes. Though executable specifications can be used as prototypes, there is again a distinction of intent which is reflected in the importance given to different attributes. Prototypes serve to explore ideas and to verify decisions. They usually generate only a part of the functionality of the future system in whatever way seems appropriate. Executable specifications, on the other side, form the basis for implementations. They must describe the complete functionality of the software system to be implemented, and they have to be as abstract as possible.

5 Summary and Conclusions

The following statements summarize my arguments for the use of executable specifications written in declarative languages.

1. Executable specifications allow to demonstrate the behavior of a software system before it is actually implemented. This has three positive consequences for software development.

> Executable components are much earlier available than in the traditional life-cycle. Therefore validation errors can be corrected immediately without incurring costly redevelopment.
>
> Requirements that are initially unclear can be clarified and completed by hands-on experience with the executable specifications.
>
> Execution of the specification supplements inspection and reasoning as means for validation. This is especially important for the validation of non-functional behavior.

2. Declarative languages, especially logic languages, combine high expressiveness with executability. They allow to write both property-oriented and model-oriented executable specifications on the required level of abstraction. Logic specification languages permit to express non-determinism in a natural way.

3. Executable specifications are constructive, i.e. they do not only demand the existence of a solution, they actually construct it.

4. Non-executable specifications which are constructive can be transformed into executable ones on almost the same level of abstraction. The resulting executable specifications are often based on search. It is not necessary to introduce new algorithms to achieve executability.

5. Executable specifications do not necessarily constrain the choice of possible implementations because only minimal design and implementation decisions are necessary to get executability. In addition, these decisions are revisable.

6. Verification of an implementation against the specification becomes superfluous if one uses the transformational approach.

6 Acknowledgements

I am grateful to I. J. Hayes and C. B. Jones whose paper inspired me to critically review the arguments that speak for and against executable specifications. I would like to thank P. Baumann and M. Fromherz for stimulating discussions, and F. Bry, P. Flener, M. Fromherz, Ch. Draxler, J. Lloyd, L. Popelinsky, R. Stadler, and G. Wiggins for helpful comments on an earlier draft of this paper.

This research has been partially supported by the Swiss National Science Foundation under the contract 2000-5.449.

7 References

[Agresti 86] W. W. Agresti (Ed.), New Paradigms in Software Development, IEEE Computer Society Press, 1986

[Bry 89] F. Bry, Logic Programming as Constructivism: A Formalization and its Application to Databases, Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 34-50, 1989

[Bundy et al. 90] A. Bundy, A. Smaill, G. Wiggins, The Synthesis of Logic Programs from Inductive Proofs, in J. W. Lloyd (Ed.), Computational Logic, Symposium Proceedings, Brussels November 1990, Springer, 1990

[Burstall, Darlington 77] R. M. Burstall, J. Darlington, A Transformation System for Developing Recursive Programs, Journal of the ACM, Vol. 24, No. 1, pp. 44-67, January 1977

[Deville 90] Y. Deville, Logic Programming, Systematic Program Development, Addison-Wesley Publishing Company, 1990

[Dijkstra 76] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976

[Flener 91] P. Flener, Towards Stepwise, Schema-Guided Synthesis of Logic Programs, in: K.-K. Lau, T. Clement (eds.), Proceedings of LOPSTR '91, Springer, 1992

[Fribourg 90] L. Fribourg, Extracting logic programs from proofs that use extended Prolog execution and induction, in: Proceedings 7th International Conference on Logic programming, MIT Press, 1990

[Fromherz 89] M. P. J. Fromherz, A Survey of Executable Specification Methodologies, Technical Report 89.05, Department of Computer Science, University of Zurich

[Fromherz 91] M. P. J. Fromherz, A Methodology for Executable Specifications - Combining Logic Programming, Object-Orientation and Visualization, PhD thesis, Department of Computer Science, University of Zurich

[Fuchs 92] N. E. Fuchs, Hoare Logic, Executable Specifications, and Logic Programs, Structured Programming 13, 1992, pp. 129-135; also: Technical Report 92.02, Department of Computer Science, University of Zurich

[Fuchs, Fromherz 91] N. E. Fuchs, M. Fromherz, Schema-Based Transformations of Logic Programs, in T. P. Clement, K.-K. Lau (eds.), Logic Program Synthesis and Transformations, Proceedings of LOPSTR '91, University of Manchester, Springer Verlag, 1992

[Gardner, Shepherdson 89] P. A. Gardner, J. C. Shepherdson, Unfold/Fold Transformations of Logic Programs, PM-89-01, School of Mathematics, University of Bristol, 1989

[Gehani 86] N. Gehani, Specifications: Formal and Informal – A Case Study, Software - Practice and Experience, 12, pp. 433-444, 1982; reprinted in [Gehani, McGettrick 86]

[Gehani, McGettrick 86] N. Gehani, A. D. McGettrick (Eds.), Software Specification Techniques, Addison-Wesley Publishing Company, 1986

[Ghezzi et al. 90] C. Ghezzi, D. Madrioli, A. Morzenti, TRIO: A Language for Executable Specifications of Real-Time Systems, Journal of Systems and Software, 12, pp. 107-123, 1990

[Goguen 86] J. A. Goguen, One, None, A Hundred Thousand Specification Languages, in: H.-J. Kugler (Ed.), Information Processing 86 (IFIP), Elsevier Science Publisher, pp. 995-1003, 1986

[Hayes, Jones 89] I. J. Hayes, C. B. Jones, Specifications are not (necessarily) executable, Software Engineering Journal, Vol. 4, No. 6, pp. 330-338, November 1989

[Hill, Lloyd 91] P. M. Hill, J. W. Lloyd, The Gödel Report, TR-91-02, Computer Science Department, University of Bristol, March 1991

[Hoare 87] C. A. R. Hoare, An overview of some formal methods for program design, IEEE Computer, 20(9), pp. 85-91, September 1987; reprinted in C. A. R. Hoare, C. B. Jones (ed.), Essays in Computing Science, Prentice-Hall, 1989

[Kemmerer et al. 87] R. A. Kemmerer, S. White, A. Mili, N. Davis, Problem Set for the Fourth Int. Workshop on Software Specification and Design, in: IEEE, ACM, Proc. Fourth Int. Workshop on Software Specification and Design, Monterey, CA, April 1987, pp. ix-x

[Kowalski 85] R. A. Kowalski, The relation between logic programming and logic specification, in: C. A. R. Hoare, J. C. Shepherdson (Eds.), Mathematical Logic and Programming Languages, Prentice-Hall International, 1985

[Lau, Prestwich 91] K.-K. Lau, S. D. Prestwich, Synthesis of a Family of Recursive Sorting Procedures, in: V. Saraswat, K. Ueda (eds.), Proceedings of the 1991 International Symposium on Logic Programming, MIT Press, 1991

[Levi 86] G. Levi, New Research Directions in Logic Specification Languages, in: H.-J. Kugler (Ed.), Information Processing 86 (IFIP), Elsevier Science Publisher, pp. 1005-1008, 1986

[Lloyd 87] J. W. Lloyd, Declarative Error Diagnosis, New Generation Computing, Vol. 5, No. 2, pp. 133-154, 1987

[Lloyd, Topor 84] J. W. Lloyd, R. W. Topor, Making Prolog More Expressive, Journal of Logic Programming, Vol. 1, No. 3, pp. 225-240, 1984

[Milner et al. 90] R. Milner, M. Tofte, R. Harper, The Definition of Standard ML, MIT Press, 1990

[O'Keefe 90] R. A. O'Keefe, The Craft of Prolog, MIT Press, 1990

[Partsch 90] H. A. Partsch, Specification and Transformation of Programs, A Formal Approach to Software Development, Springer Verlag, 1990

[Quine 70] W. V. O. Quine, Philosophy of Logic, Prentice-Hall, 1970.

[Stadler 90] R. Stadler, Ausführbare Beschreibung von Directory-Systemen - Prolog-basierte Spezifikation der Architektur von Directory-Systemen, PhD thesis, Department of Computer Science, University of Zurich; also to appear in the Informatik-Fachberichte series, Springer, 1991)

[Sterling, Shapiro 86] L. Sterling, E. Shapiro, The Art of Prolog, Advanced Programming Techniques, MIT Press, 1986

[Stolze et al. 91] M. Stolze, M. Gutknecht, R. Pfeifer, Integrated Knowledge Acquisition: Toward Adaptive Expert System Design, Technical Report 91.04, Department of Computer Science, University of Zurich

[Tamaki, Sato 84] H. Tamaki, T. Sato, Unfold/Fold Transformation of Logic Programs, Proc. of the Second Internatl. Conference on Logic Programming, S.-Å. Tärnlund (Ed.), pp. 127-138, 1984

[Terwilliger, Campbell 89] R. B. Terwilliger, R. H. Campbell, PLEASE: Executable Specifications for Incremental Software Development, Journal of Systems and Software, 10, pp. 97-112, 1989

[Terwilliger 90] R. B. Terwilliger, An Overview and Bibliography of ENCOMPASS, an Environment for Incremental Software Development Using Executable, Logic-Based Specifications, ACM SIGSOFT, Software Engineering Notes, Vol. 15, No. 1, pp. 93-94, January 1990

[Turner 85] D. A. Turner, Functional programs as executable specifications, in: C. A. R. Hoare, J. C. Shepherdson (Eds.), Mathematical Logic and Programming Languages, Prentice-Hall International, 1985

[Ural 90] H. Ural, Specifications of Distributed Systems in Prolog, Journal of Systems and Software, 11, pp. 143-154, 1990

[Wing 90] J. M. Wing, A Specifier's Introduction to Formal Methods, IEEE Computer, Vol. 7, No. 5, pp. 8-24, September 1990

[Zave 82] P. Zave, An Operational Approach to Requirements Specification for Embedded Systems, IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, pp. 250-269, May 1982; reprinted in [Agresti 86] and [Gehani, McGettrick 86]

[Zave, Yeh 81] P. Zave, R. T. Yeh, Executable Requirements Specification for Embedded Systems, Proceedings of the 5th International Conference on Software Engineering, pp. 295-304, San Diego, 1981; reprinted in [Gehani, McGettrick 86]