PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 41, NO. 2, PP. 85–100 (1997)

AN EXECUTABLE SPECIFICATION FORMALISM REPRESENTING ABSTRACT DATA TYPES

József BOROVEN

Department of Telecommunication
Technical University of Budapest
H–1521 Budapest, Hungary
Phone: +36 1 463-1613

Received: January 9, 1995

Abstract

It has been proved to be very useful and necessary to give formal specifications of software systems to be developed. The specifications should help to avoid the necessity of creating *prototypes* by offering *direct executability*. A useful specification language aiming the description of abstract data types – while maintaining *abstractness* – should also support the representation of *states* of objects, as well as support the *transformation* of declarative specifications into efficiently executable code.

The present paper is intended to give an informal description of a specification language aimed to offer the features discussed above. Although the development of the language has mainly been motivated by the object-oriented language (OMOHUNDRO, 1993), it is intended to function as a specification formalism at a much broader field.

Introduction

Specification is a crucial phase in the software life-cycle. One of the reasons for that is the error-proneness of this part, according to (BOEHM, 1979) over 60 percent of the errors uncovered in software systems were due to shortcomings in the specifications themselves.

Probably the best solution for the above mentioned problem is to make specifications *executable*, thus allowing the end users and developers to uncover the errors in an earlier phase of software development.

Data abstraction is a very important concept and method in formal specification techniques as well as in up-to-date software methodologies, especially in object-oriented programming. Nowadays there are basically two approaches for the specification of abstract data types:

– *Algebraic specification method:* (see e.g. SPIVEY, 1989) In this approach data objects are characterized by the operations of the data type, and the semantics of operations are defined by algebraic equations. No representation of the data type is given.
– *Constructive specification method:* (see e.g. BJORNERD and JONES, 1980) This technique uses already existing building blocks (called

meta types) for building a *model* of the abstract data type to be specified. The operations of the data type are expressed in terms of the operations of the meta types.

Both methods have their advantages and disadvantages. Algebraic methods are more abstract, more general, free of unnecessary implementational details. On the other hand, they are sometimes more difficult to construct than constructive specifications, *sometimes it is even impossible to specify in the algebraic approach* (BOEHM, 1979). But the most serious problem with this method is its inability to incorporate the notion of *state*[1]

The constructive specification method is easier to learn (especially for practical programmers). It can represent states in a natural way. The main problems with constructive specifications are: *overspecification* and *lack of abstraction.*

In this paper a specification language is described which incorporates both methods mentioned above.

Overview of Existing Specification Methods

As it was mentioned before the existing formalisms can be partitioned into two main groups: algebraic and constructive specifications.

One of the most well-known representatives of the algebraic approach is the *CIP–L* language introduced by the Munich Project CIP (SPIVEY, 1989). The language is a *wide-spectrum language*, with *applicative* as well as *imperative features*, supported by a *program transformation system.* Drawback of the language is, beside the general problems with the inability with incorporating states, and difficulty with creating such specifications, that it is *not executable.*

OBJ is an executable algebraic specification language, it can also be regarded as a *functional programming language* augmented with *abstract data types* (ADTs) (GOGUEN and MESEGUER, 1984). The general problems of algebraic specifications hold.

EQLOG (GOGUEN and MESEGUER, 1984) incorporates logic and functional programming with the notion of ADTs into one semantically coherent framework. Unfortunately – because of the algebraic approach – the general problems of that method is applied. On the other hand – because of the purely declarative syntax given – no implementation has been developed for the language yet.

Efforts have been made to *incorporate the notion of state into algebraic methods.* In *COLD* (BRIL, 1991) state transitions are represented as

[1]The only exception is the COLD specification language (BRIL, 1991), but it can represent state transitions only via the changing of the underlying algebra, which doesn't seem to be the most appropriate method imaginable.

changes in the algebraic structure representing the ADT to be specified. It seems that this approach doesn't lead closer to specifications leading to efficient implementations.

As the most prominent formalism using the constructive approach we take *VDM* (JONES, 1982, 1986). Because of the method used there is no problem with *states*, it supports the concept of *stepwise refinement*. The drawbacks of the formalism are: it is basically a *paper and pencil method*, has no direct support for mechanical transformation. It has *no parameterizable classes it is not executable*. Similar remarks apply also to *Z*.

There have been some efforts towards *combining the algebraic and constructive specification methods*. The advantages of a language having the features of both formalisms are obvious. The most remarkable proposals are the following:

Larch (HORNING, 1985) includes a common *shared language* and a number of interface languages. There seem to exist two basic problems with the formalism:

- *Traits* (modules in Larch) are not ADTs. The semantics of them are context-dependent.
- The interface languages are programming language-dependent. It makes the specifications to be language-dependent, more difficult to follow.

NUSL (JIANG XU, 1988) is also a language which combines the two main approaches. Unfortunately it does not support the incorporation of *state* into constructive specifications.

RAISE contains all the components of the two methods augmented with the possibility of describing *processes*, and writing specification layers at an imperative level. Unfortunately it has two deficiencies: it is *not executable* and *does not really support the introduction of states into specifications.*

Motivation and Objectives

The main ideas of the specification language to be introduced are the following:

- Since objects have *states*, the specification of ADTs must incorporate the notion of states. It leads to *constructive specification methods.*
- The incorporation of the *algebraic specification method*, for the following reasons: it provides a more abstract level of specification, can serve as *requirement theories* (type bounds), and for an object-oriented language which has the concept of *abstract classes* (e.g. Sather) there seem to be no other way to model them.

- As it directly follows from the two points above, the language should have as a feature *the combination of algebraic and constructive specification methods.* Since both methods have their advantages, and in certain situations the use of one of them is much more suitable and natural than the other, it should be allowed to have the opportunity to construct *mixed-method specifications.*
- The language must support the *stepwise refinement* process for constructing specifications. It means that especially the part of specification made in the constructive way should be allowed (and supported) to be combined from parts created using different approaches (namely *predicative* and *functional*), supporting *consistency checking* with the algebraic (abstract) and constructively defined ADT phases. The process of stepwise refinement should be supported by a (partly) intuition-driven transformation system.
- The *introduction of states* into the specification should be supported by the language. It is possible (of course) only with the constructive specification phase.
- The language must be *executable* in order to support the remedy of deficiencies of specifications at an early phase of software development.

Stepwise Refinement

The specification of an abstract data type using the language to be presented can be approached in the following ways:

- Algebraic specification
- Model-oriented (constructive specification)

The *constructive part* can be broken down according to the following approaches:

- The method used for specification. It can be:
  - *implicit* (or in other terms: predicative): The well-known pre- and postcondition technique (see e.g. (JONKERS, 1991)
  - *explicit* (or in other words: functional): Pure functional specification (see e.g. (HENDERSON, 1986).

The explicit way is more algorithmic, more specific than the other method.

- The layer of the ADT to be specified
  - *applicative layer*: The specification of the functions of the ADT without side-effects (no state (transition) involved)
  - *imperative layer*: The definition of state transitions by functions of the ADTs as their *side effects*.

In the majority of cases the process of ADT development can be outlined in the following way:

1. An *algebraic specification* of the ADT is defined.
2. The *applicative layer* of the ADT is defined. First an appropriate *abstract representation* of the ADT has to be specified. In order to avoid the overspecification, the implicit specification should be defined first. The explicit specification can be skipped.
3. The *imperative layer* of the ADT is specified. The sequence for defining in the implicit and explicit way should be the same.

It should be mentioned that the construction of the algebraic specification is not always feasible and sometimes can be omitted.

Overview of the Language

In the forthcoming the fundamental concepts of the language will be presented. The (informal) syntax and semantics will be shown via examples.

Type Signature

The signature of an ADT is the most standing part of the development process. It contains the enlisting of *sorts* (presently the number of sorts introduced by and ADT is limited to one), and *functions* with their functionalists. The type signature serves purely syntactical purposes, namely the intended use of it is *textual inclusion* into the parts representing the further phases of development. It could be augmented by *comments* to describe the intended meaning of the operators.

The type signature is the only representation of the ADT which is not executable.

Here and in the forthcoming discussion the features of the language are demonstrated via the well-known *stack* ADT.

(The per cent symbol denotes *comments*.)

```
TYPESIG stack_sig     %%% TYPE SIGNATURE
SORT Stack(Elem)      % Parameterizable Classes
FUNCT
create:               →Stack
is_empty:             Stack → BOOL
push:                 Stack,Elem → Stack
pop:                  Stack → Stack
top:                  Stack → Elem
END stack_sig
```

Algebraic Specification

This part gives semantics to type signatures by defining a (many-sorted) algebra via axioms. The language of the axioms is *Horn clause logic with equality.* This part is executable using the procedural semantics of equality-based logic programming (*narrowing*).

This method is unable to incorporate *states.* On the other hand an algebraic specification has an important role in the development process, for the following applications:

- *Ultimate reference point.* The other (constructive) parts of the specification can be tested against it, checking whether they satisfy the axioms of the algebraic description.
- It can function as a *requirement theory* for implementing *type bounds.*
- It is fully equivalent with the concept of an *abstract class* in object-oriented languages (e.g. Sather).

It should be mentioned that in several cases it is difficult, unnatural or even impossible to construct an algebraic specification for an ADT (see the example of the finite state machine later).

```
ALG alg_stack     %"pure" ALGEBRAIC spec.  with NO STATE

USETYPESIG stack_sig

AXIOM
is_empty(create)=TRUE;
is_empty(push(s,e))=FALSE;
pop(empty)=ERROR;
pop(push(s,e))=s;
top(empty)=ERROR;
top(push(s,e))=e;

END alg_stack
```

Building Blocks of the Constructive Method

The predefined meta types (building blocks) of the constructive method have been borrowed from VDM. They can be regarded as abstract data types with firm mathematical properties. They are the following:

- SET:
  Set type with the usual operators: union, intersection etc.
- SEQ:
  List type with concatenation etc.

- MAP:
  Finite function with function composition etc.
  For details refer to (JONES, 1986).

Since our language supports *mixed language specifications, it is allowed to use already existing ADT specifications in model construction.* These abstract data types can even be specified algebraically. But it should be mentioned that such mixed method specifications can only be used for the construction of the applicative layer of a specification, since *the presence of algebraic modules* makes the interpretation of *states* difficult. So the primary application area of mixed method specifications is the field of *fast prototyping.*

Implicit Constructive Specification without State

The least specific constructive specification. It contains the *abstract representation of the ADT* and the predicative definitions of functions in the form of *pre- and postconditions.*

This kind of specification can only be used for specifying 'pure' functions (functions without side effects). No state modification can be specified, although the abstract representation itself should be regarded as the data type of an object the valuation function of which represents the state.

The language of logic formulas representing the pre- and postconditions should be *Horn clause logic.* As the procedural semantics of the language it seems to be reasonable to choose SLDNF resolution, assuming Prolog's computation rule and SLDNF tree traversal strategy in order to get a reasonably efficient implementation (although it should be mentioned that on the other hand it means a serious compromise, because of the incompleteness of this procedural semantics).

Interpretation of the pre- and postcondition notation:

Function signature:

```
func:  <arguments> → <result>
```

Signature of predicative specification:

```
PRE-func(<arguments>).
POST-func(<arguments>,<result>).
```

Semantics of pre-postcondition specification:

```
op-func(<arguments>,<result>) ←
        pre-op(<arguments>),     %type-constraint for arguments
        post-op(<arguments>,<result>).
```

```
CON pred_stack;

USETYPESIG stack_sig
```

```
ABSREP
Stack=SEQ(Elem);

PREPOST
%one possible functional spec.  appears as comment
%create()=<>;
PRE-create.
POST-create(<>).

%is_empty(s)=(s=<>);
PRE-is_empty(s).
POST-is_empty(s,s=<>).

%push(s,e)=<e>||s;
PRE-push(s,e).
POST-push(s,e,<e>||s).

%pop(s)= IF is_empty(s) THEN ERROR ELSE TL s FI;
PRE-pop(s):- not is_empty(s).
POST-pop(s,TL s).

%top(s)= IF is_empty(s) THEN ERROR ELSE HD s FI;
PRE-top(s):- not is_empty(s).
POST-top(s,HD s).

END pred_stack
```

Explicit Constructive Specification without State

This part is also aimed at only building the applicative layer of the software description, using a *pure functional language without side effects* and state.

The main difference between this and the previous method is that this kind of specification is more specific than the previous one, as it is well-known there are a number of functional definitions corresponding to a logic definition (according to the relationship between functions and relations). A functional description is always closer to imperative, algorithmic programming.

On the other hand it is generally reasonable to *postpone decisions* which make the specification more specific as far as possible in the software development process in order to avoid overspecification.

The procedural semantics of functional specifications is *functional rewriting.*

```
CON fun_stack;

USETYPESIG stack_sig
```

```
ABSREP
Stack=SEQ(Elem);

FUNCTION
create( )=< >;
is_empty(s)=(s=< >);
push(s,e)=<e>||s;
pop(s)= IF is_empty(s) THEN ERROR ELSE TL s FI;
top(s)= IF is_empty(s) THEN ERROR ELSE HD s FI;

END fun_stack
```

Implicit Constructive Specification with State

This part introduces the notion of *state* via the creation of an object of the type according to the abstract representation. The state is interpreted as the valuation function of this object. The specification appearing in this part is responsible for the description of state transitions initiated by the functions of the ADT (in other terms: side effects), thus extending the applicative specification into an imperative one.

Interpretation of the notation used:

Function signature:

```
func:  <arguments> → <result>
```

Signature of predicative specification:

```
PRE-func(<old-state>,<arguments>)
POST-func(<old-state>,<arguments>,<new-state>)
```

Semantics of pre-postcondition specification:

```
state-op-func(<old-state>,<arguments>,<new-state>) ←
          pre-op(<old-state>,<arguments>),
          %type plus state constraint
          post-op(<old-state>,<arguments>,<new-state>).
          %state transformation
```

Comments related to the language of logic used for stateless predicative specifications also apply here.

```
CONSTATE predst_stack       %IMPLICIT CONSTRUCTIVE spec.:
                            %the STATE spec.  part
                            % "abstract" IMPLEMENTATION CLASS

USECON fun_stack            %it could have been "pred_stack" as well

STREP                       %state representation
```

```
v:Stack                     %Stack=SEQ(Elem), defined in the "CON" class
                            % "v" is a variable belonging to type Stack

PREPOSTS
%create:           →Stack;
PRE-create(s).  %any state is acceptable
POST-create(s,< >).

%is_empty:         Stack → BOOL
PRE-is_empty(s,s).
POST-is_empty(s,s,s).

%push:             Stack,Elem → Stack
PRE-push(s,s,e).
POST-push(s,s,e,<e>||s).

%pop:              Stack → Stack
PRE-pop(s,s):- not is_empty(s).
POST-pop(s,s,TL s).

%top:              Stack → Elem
PRE-top(s,s):- not is_empty(s).
POST-top(s,s,s).

END predst_stack
```

Explicit Constructive Specification with State

Used for the same function as the previous part, the difference is in the more explicit method used for specification.

The comments made in connection with explicit stateless specifications also apply here.

The present combination of specifications of the applicative and imperative layer represents the most concrete kind of an abstract implementation class, which is closest to an implementation using an imperative language.

Interpretation of the functional notation:

Function signature:

`func:  <arguments> → <result>`

Signature of function representing state transition:

```
ST-func(<old-state>,<arguments>)= <new-state>
CONSTATE funst_stack         %EXPLICIT CONSTRUCTIVE spec.:
                             %the STATE trans.  spec.  part
                             % "abstract" IMPLEMENTATION CLASS
```

```
USECON fun_stack          %it could have been "pred_stack" as well

STREP
v:Stack

FUNCTION

%create:                  →Stack;
ST-create(s)=< >;         %new state :  < >

%is_empty:                Stack → BOOL
ST-is_empty(s,s)=s;       %no state change

%push:                    Stack,Elem → Stack
ST-push(s,s,e)=<e>| |s;

%pop:                     Stack → Stack
ST-pop(s,s)= IF is_empty(s) THEN ERROR ELSE TL s FI;
%partial function !

%top:                     Stack → Elem
ST-top(s,s)= IF is_empty(s) THEN ERROR ELSE s FI;
%partial function !

END funst_stack
```

Introducing 'Object-Oriented Notation'

In object-oriented languages the notation used for function applications is different from the one that was used before. For example:

```
    s.push(e)
```

is used instead of:

```
    push(s,e).
```

It represents the notion that the operators belong to and operate on the object and *take the object as their implicit first parameter.* This notation – while should be allowed – should be regarded as a *syntactic sugar* for the latter.

The semantics of operations appearing in the 'object-oriented' form are expressed in terms of operations appearing in the original form using transformational semantics. The transformation can be described by a translation schema. SELF refers to the implicit first parameter of the function:

```
TYPESIG stack_self %type signature for "object-oriented" notation
```

```
SORT Stack(Elem)

     FUNCT
     create:                 →Stack            ::      →Stack
     is_empty:               Stack → BOOL      ::      SELF → BOOL
     push:                   Stack,Elem → Stack        SELF,Elem → Stack
     pop:                    Stack → Stack     ::      SELF → Stack
     top:                    Stack → Elem      ::      SELF → Elem

     END stack_self
```

Specification of a Finite State Automaton: An Example

The above presented specification sequence from algebraic to explicit imperative constructive specifications is not always so easy to construct. As a counter-example, let us take the specification of a finite automaton.

At first sight the task seems to be unfeasible using the algebraic way. On the other hand the constructive method offers an easy way to success:

The first step is to create the type signature:

```
TYPESIG finaut_sig

SORT FinAut
FUNCT
create:          FinAut →FinAut
is_in_final:     FinAut → BOOL
trans:           FinAut,CHAR →FinAut

END finaut_sig
```

The next one is to find an appropriate *abstract representation*:

```
ABSREP
FinAut ::        {                  %composite type
                 AllStates          :SET(State);
                 InitState          :State;
                 FinalStates        :SET(State);
                 Alphabet           :SET(CHAR);
                 Delta              :MAP((State,CHAR) → State);
                 ActState           :State %actual state
                 }
State ::         INT      %simplest choice
```

A functional specification of the applicative layer:

```
CON fun_finaut;
```

```
USETYPESIG finaut_sig
ABSREP
FinAut ::       {
                AllStates       :SET(State);
                InitState       :State;
                FinalStates     :SET(State);
                Alphabet        :SET(CHAR);
                Delta           :MAP((State,CHAR) → State);
                ActState        :State
                }
State ::        INT

FUNCTION
create(F) = {   AllStates       = F.AllStates;
                InitState       = F.InitState;
                FinalStates     = F.FinalStates;
                Alphabet        = F.Alphabet;
                Delta           = F.Delta;
                ActState        = F.InitState } % !

is_in_final(F) = (F.ActState member_of F.FinalStates)

trans(F,C) = {  AllStates       = F.AllStates;
                InitState       = F.InitState;
                FinalStates     = F.FinalStates;
                Alphabet        = F.Alphabet;
                Delta           = F.Delta;
                ActState        = F.Delta(F.ActState,C) }
END fun_finaut
```

As for *state representation* it is reasonable to choose the *ActState* part of the abstract representation (the other parts are obviously constants):

```
STREP
a:FinAut.ActState
```

Using the state representation chosen a possible functional specification of the imperative layer can be the following:

```
CONSTATE funst_finaut

USECON fun_finaut

STREP
a:FinAut.ActState

FUNCTION
```

```
    ST-create(F.ActState,F) = F.InitState

    ST-is_in_final(F.ActState,F) = F.ActState

    ST-trans(F.ActState,F,C) = F.Delta(F.ActState,C)

    END funst_finaut
```

(The predicative specification part – since they should not cause any trouble – is intentionally left out.)

In fact, the algebraic specification of the ADT is not impossible (see e.g. (SPIVEY, 1989), although very complicated and very unnatural compared to the above shown constructive method. The reason for that is obvious: the constraints represented by the model can only be expressed via the extensive use of inheritance, which (in this case) makes the solution rather intractable.

Inheritance

Two kinds of inheritance mechanisms are provided (adopted) from (SPIVEY, 1989):

- BASED_ON : The constituents of the inherited type are available after the BASED_ON clause. Multiple inheritance is possible. The semantics are *not based on textual substitution*[2]. Identifiers having the same names belonging to different base classes can be reached via *quantification*.
- INCLUDE : Specifies a *renaming* of a class followed by inclusion. Referential transparency is still maintained. Typical example: (constructive) specification of a stack based on lists (for details, refer to (SPIVEY, 1989).

Problems, Possible Further Improvements

- *Precise syntax* must be given for the language. It is obviously a minor problem.
- A *general semantics framework* for the mixed-method specification language should be given. One way seems to be feasible at first sight: *a transformational plus denotational semantics*. On the other hand a *declarative semantics* should be defined for the applicative layer of

[2]One of the most serious drawbacks of the Sather language is that inheritance is completely based on *textual substitution*, in such a way which seriously endangers referential transparency.

the language (related problem: the *integration of functional and logic programming*).

- The system should support the checking of the *orthogonality* of specifications, although in some practical applications it is not one of the topmost requirements against specifications.

As an extension of the application of algebraic specification modules (beside the use of them as *abstract classes*, they can be considered as *requirement theories*, which can be used for the implementation of *type bounds*. The type bounds can be checked by an automated theorem prover.

Conclusion

A framework of an executable specification language for developing and testing abstract data types has been presented. Although the language is not in the phase of implementation, and still has some basic problems to be solved at the field of theoretical background, it hopefully will be able to be used at the following main fields:

- to function as a (more or less) *wide-spectrum* specification language for (general) object-oriented languages
- because of its executable nature, to serve as an object-oriented functional-logic programming language (considering the applicative layer)

References

ANDREWS, D.: Data Reification and Program Decomposition. In D. Bjorner, C. B. Jones, M. Mac an Airchinnigh, and E. J. Neuhold, editors, *VDM'87 : VDM – A Formal Method at Work*, number 252 in Lecture Notes in Computer Science, pp. 389–422. Springer-Verlag, 1987.

BJORNER, D. – JONES, C. B.: Formal Specification and Software Development. Prentice-Hall International Series in Computer Science. Prentice-Hall International, first edition, 1982.

BOEHM, B. K. Software Engineering: R and D trends and Defense Needs. MIT Press, 1979.

BRIL, R. J. : A Model-oriented Method for Algebraic Specifications Using Cold-1 as Notation. In S. Prehn and W. J. Toetenel, editors, *VDM'91 : VDM – Formal Software Development Methods*, No. 551 in Lecture Notes in Computer Science, pp. 106–124. Springer-Verlag, 1991.

CARDELLI, L. – WEGNER, P.: On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, Vol. 17 (4), pp. 471–522, December 1985.

DOMA, V. – NICHOLL EZ, R.: A System for Automatic Prototyping of z Specifications. In S. Prehn and W. J. Toetenel, editors, *VDM'91 : VDM – Formal Software Development Methods*, number 551 in Lecture Notes in Computer Science, pp. 189–203. Springer-Verlag, 1991.

GOGUEN, J. A. – MESEGUER, J.: Equality, Types, Modules, and (why not) Generics for Logic Programming. *Journal of Logic Programming*, Vol. 12, pp. 179–210, 1984.

GOGUEN, J. A. – MESEGUER, J.: Programming with Equalities, Subsorts, Overloading, and Parameterization in obj. *Journal of Logic Programming*, Vol. 12, pp. 257–279, 1992.

HENDERSON, P.: Functional Programming, Formal Specification, and Rapid Prototyping. *IEEE Transactions on Software Engineering*, Vol. 12(2) pp. 241–250, February 1986.

HORNING, J. J.: Combining Algebraic and Predicative Specifications in Larch. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Development : TAPSOFT*, number 186 in Lecture Notes in Computer Science, pp. 12–26. Springer-Verlag, 1985.

JAGER, M. – GLOGER, M. – KAES, S.: Sample – a Functional Language. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88 : VDM – The Way Ahead*, number 328 in Lecture Notes in Computer Science, pp. 202–217. Springer-Verlag, 1988.

JONES, C. B.: Software Development : A Rigorous Approach. Prentice-Hall International Series in Computer Science. Prentice-Hall International, first edition, 1980.

JONES, C. B.: Systematic Software Development Using VDM. Prentice-Hall International Series in Computer Science. Prentice-Hall International, first edition, 1986.

JONKERS, H. B. M.: Upgrading the Pre- and Postcondition Technique. In S. Prehn and W. J. Toetenel, editors, *VDM'91 : VDM – Formal Software Development Methods*, number 551 in Lecture Notes in Computer Science, pp. 428–456. Springer-Verlag, 1991.

JIANG, X. – XU, Y.: Nusl: An Executable Specification Language Based on Data Abstraction. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88 : VDM – The Way Ahead*, number 328 in Lecture Notes in Computer Science, pp. 124–138. Springer-Verlag, 1988.

MAC AN AIRCHINNIGH, M.: Introduction to the vdm Tutorial. In D. Bjorner, C. B. Jones, M. Mac an Airchinnigh, and E. J. Neuhold, editors, *VDM'87 : VDM – A Formal Method at Work*, number 252 in Lecture Notes in Computer Science, pp. 356–361. Springer-Verlag, 1987.

MAC AN AIRCHINNIGH, M.: Specification by Data Types. In D. Bjorner, C. B. Jones, M. Mac an Airchinnigh, and E. J. Neuhold, editors, *VDM'87 : VDM – A Formal Method at Work*, number 252 in Lecture Notes in Computer Science, pp. 362–388. Springer-Verlag, 1987.

NIELSEN, M. –HAVELUND, K.– WAGNER, K. R.–GEORGE, C.: The Raise Language, Method and Tools. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88 : VDM – The Way Ahead*, number 328 in Lecture Notes in Computer Science, pp. 376–405. Springer-Verlag, 1988.

OMOHUNDRO, S.: The sather 1.0 specification. Technical Report, The International Computer Science Institute, 1947 Center St, Suite 600, Berkeley, CA 94704, February 1993.

PARTSCH, H. A.: Specification and Transformation of Programs : A Formal Approach to Software Development. Texts and Monographs in Computer Science. Springer-Verlag, first edition, 1990.

POTTER, B. – SINCLAIR, J. – TILL, D.: An Introduction to Formal Specification and Z. Prentice-Hall International Series in Computer Science. Prentice-Hall International, first edition, 1991.

SPIVEY, J. M.: Understanding Z : A Specification Language and its Formal Semantics. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, first edition, 1989.