PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 13/94

Logical Specifications: 1. Introduction and Overview

Thomas S. E. Maibaum
Paulo A. S. Veloso

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 13/94 ISSN 0103-9741
Editor: Carlos J. P. Lucena June, 1994

Logical Specifications: 1. Introduction and Overview *

Thomas S. E. Maibaum †

Paulo A. S. Veloso

* Research partly sponsored by the Brazilian agencies CNPq, RHAE and FAPERJ, British agency SERC and European Community agencies.

† Dept. of Computing, Imperial College of Science, Technology and Medicine, Lodon, UK.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

LOGICAL SPECIFICATIONS: 1. INTRODUCTION AND OVERVIEW

Thomas S. E. MAIBAUM and **Paulo A. S. VELOSO**
{e-mail: tsem@doc.ic.ac.uk and veloso@inf.puc-rio.br}

PUCRioInf MCC 13/94

Abstract. The logical approach to formal specifications regards specifications as axiomatic presentations of theories in standard first-order logic. Its motivations come mainly from two related sources: concepts akin to program verification and 'liberal' specifications (which provide flexibility for specifying what one wishes without forcing over-specification). The rationale for our standpoint is the realisation that programming has to do with syntactical objects rather than with their semantical counterparts and, thus, that program and specification development consists of the manipulation of theories. This introductory report is intended to set the stage and provide a general overview. It presents and discusses some general ideas underlying our property-oriented approach. We start by examining the motivations for our approach and then proceed to consider some of its consequences for stepwise development: implementations and their composition as well as parameterisation and instantiation. We also briefly comment on our approach to errors via liberal specifications. This report is a draft of the first section of a handbook chapter. Subsequent reports will analyse these and related points in more detail.

Key words: Formal specifications, program development, logical approach, logical theories, axiomatic presentations, liberal specifications.

Resumo. O enfoque lógico para especificações formais trata especificações como apresentações axiomáticas de teorias na lógica usual de primeira ordem. Suas motivações vêm principalmente de duas fontes relacionadas: conceitos próximos à verificação de programas e especificações 'liberais' (as quais fornecem flexibilidade para se especificar o que se deseja sem forçar super-especificação). A razão para nosso posicionamento é a observação que programação tem a ver com objetos sintáticos e não com suas contrapartidas semânticas e que, portanto, desenvolvimento de programas e especificações consiste de manipulações de teorias. Este relatório introdutório pretende fornecer um pano de fundo através de uma visão panorâmica, apresentando e discutindo algumas idéias gerais subjacentes a nosso enfoque orientado a propriedades. Começamos examinando as motivações para o enfoque e considerando algumas de suas conseqüências para o desenvolvimento por etapas: implementações e sua composição bem como parametrização e instanciação, também comentando brevemente sobre nosso enfoque para erros. Este relatório é um esboço da primeira seção de um capítulo de um manual. Relatórios subseqüentes analisarão esses pontos, e outros relacionados, em mais detalhe.

Palavras chave: Especificações formais, desenvolvimento de programas, enfoque lógico, apresentações axiomáticas, teorias lógicas, especificações liberais.

NOTE

This report is a draft of the first section of a chapter in a forthcoming volume of the Handbook of Logic in Computer Science

Other reports, corresponding to the remaining sections, are in preparation. The plan of the chapter - and series of reports Logical Specifications - is as follows.

1. Introduction and Overview
2. Specifications as Presentations
3. Extensions of Specifications
4. Interpretation of Specifications
5. Implementation of Specifications
6. Parameterised Specifications
7. Conclusion: Retrospect and Prospects.

The chapter - and series of reports - is intended to provide an account of the logical approach to formal specification development.

Any comments or criticisms will be greatly appreciated.

The next report in this series is planned to be Logical Specifications: 2. Specifications as Presentations, with the following contents

2 SPECIFICATIONS AS PRESENTATIONS

2.1 Introduction

2.2 Languages

2.3 Structures and models

2.4 Theories, presentations and specifications

2.5 Inductive sorts and specifications

2.6 Errors and underdetermined operations

2.7 Reasoning about programs

2.8 Liberally constrained specifications

2.9 Example specifications

ACKNOWLEDGEMENTS

Research reported herein is part of an on-going research project. Partial financial support from British, European Community and Brazilian agencies is gratefully acknowledged. The hospitality and support of the institutions involved have been very helpful. Collaboration with Martin R. Sadler, Sheila R. M. Veloso and José L. Fiadeiro was instrumental in sharpening many ideas. The authors would like to thank the following for many fruitful discussions on these and related topics: Carlos J. P. de Lucena, Samit Khosla, Atendolfo Pereda Bórquez, Douglas R. Smith, Haydée W. Poubel and M. Claudia Meré. Special thanks go to Tarcísio H. C. Pequeno and Roberto Lins de Carvalho

CONTENTS *

List of Figures

* See the preceding note for an explanation of the numbering system.

1 Introduction and Overview

This chapter[1] presents a logical approach to formal specifications motivated by its usage in program development. This approach is based on regarding specifications as axiomatic presentations of theories in first-order logic. We shall use familiar logical concepts and terminology, but we shall generally emphasise a constructive approach: constructions that are guaranteed to provide the desired results, rather than a-posteriori verification.

This introductory section is intended to set the stage and provide a general overview. It presents and discusses some general ideas underlying our property-oriented approach.

We start by examining the motivations for our approach, which is based on viewing programs and specifications as syntactical, rather than semantical, objects. We then proceed to examine some consequences of this viewpoint for stepwise development: implementations and their composition as well as parameterisation and instantiation. We then briefly comment on our approach to errors via liberal specifications. We finally outline the contents of the subsequent sections[2] of this chapter, where these and related points will be examined in more detail.

1.1 The logical approach

The logical approach to formal specifications regards specifications as axiomatic presentations of theories in standard first-order logic. Thus we use the full expressive power of first-order formulae, in contrast to other approaches that rely on fragments, such as (conditional) equations. Also, we emphasise a language-oriented viewpoint, rather than a model-oriented one (which generally relies on some special kinds of models, such as finitely generated models).

The motivations for the logical approach to formal specifications come mainly from two related sources (Maibaum *et al.* 1991). On the one hand, logical axioms employ language and concepts akin to program verification (Manna 19740); on the other hand, the logical formalism accommodates 'liberal' specifications, which provide flexibility for specifying what one wishes without forcing over-specification (Maibaum and Veloso 1981; Maibaum and Turski 1984; Veloso *et al.* 1985; Maibaum 1986). In addition, this logical approach has been instrumental in extending some of these ideas to problem solving (Veloso and Veloso 1981; Veloso 1984,1988) and to formal algorithm design (Smith 1985, 1990, 1992).

---

[1] See the preceding note for an explanation of the terminology 'chapter', 'section', etc., as well as for the numbering system.

[2] The susbsequent sections are planned to be issued as reports in this series.

This approach comes from the tenet that program and specification development consists of the manipulation of theories. This assertion can be taken as a fact or as a viewpoint. The purpose of this introductory section is to explain this position. Some arguments will be oriented towards convincing of the truthfulness of this fact, others will try to show the benefits of this viewpoint. The main point is that viewing the process of program development as manipulation of theories does shed some light on issues in stepwise development of programs, with accompanying benefits.

1.2 Programs and theories[1]

The rationale for our standpoint is the realisation that programming has to do with syntactical objects rather than with their semantical counterparts. That a program is a syntactical entity hardly needs any elaboration: a program text is a syntactical description of the transformation effected by it. This viewpoint is further stressed by viewing an abstract program manipulating an abstract data type as a program schema (Manna 1974). The very purpose of a program is to transform input data into output results. But in so doing, a program does not manipulate directly real-world objects, only their symbolic representations (Ledgard and Taylor 1977): a program to compute, say, the greatest common divisor will actually manipulate numerals representing numbers rather than numbers themselves.

Let us see how these considerations fit into the process of program development. The starting point is the specification of the desired input-output behaviour, which is given by syntactical entities, like first-order formulae. The product consists of (the text of) an abstract program. Now, the proof that the program does satisfy the given specification amounts to syntactical manipulations employing, say, verification conditions (Manna 1974). The process of deriving a program from a specification is even more clearly one of syntactical transformations.

Notice that in reasoning about the behaviour of a program, say in proving the verification conditions, one must rely on some knowledge concerning the objects manipulated by it. But such knowledge can, and should be, at the appropriate level of abstraction. Of course, the very idea of data abstraction involves hiding representation details. Conceptually this means dealing with abstract structures or, more aptly, with their behaviour. And this is the point we wish to stress: the important aspects of the behaviour of these abstract structures can be captured by axioms, which are syntactical descriptions of their relevant properties.

---

[1] More details on these points are planned to appear in the forthcoming report Logical Specifications: 2. Specifications as Presentations.

So, programs and the objects manipulated by them are actually syntactical entities described by their properties, as captured, say, by axioms. But, what about the programs themselves, or rather their behaviour? Programming languages, in addition to syntax, do have semantics. But, notice that the so-called axiomatic semantics, à la Hoare, consist of axioms and rules of inference. And even the denotational methods give description of functions, rather than functions proper.

Thus, our view is that the process of program development can be described, explained and understood - and more fruitfully so - entirely by syntactico-axiomatic means, without recourse to other entities. In connection with this process two points are worth mentioning. Firstly, the process can involve refinement steps, and we will come to this shortly. Secondly, it does involve some "creativity" in taking good design decisions and we are not claiming to automate this. What we wish to stress is the feature that these design decisions can be recorded in formulae, which participate in the process, a point that will be elaborated upon in the sequel.

1.3 Implementations and theories[1]

Let us now consider program development by stepwise refinements. Here one postulates some abstract data type (ADT), suitable for the problem at hand, which has to be implemented on the available system. The end product consists of (the text of) an abstract program manipulating the postulated ADT, together with a suite of (texts of) modules implementing the ADT's on more concrete ones until reaching the available level.

Now one needs some knowledge about the relevant properties of the abstractions involved. This is provided by the axioms in the specifications of the ADT's. The proof that the abstract program does exhibit the required behaviour consists, as before, of syntactical manipulations that derive the verification conditions from the ADT specification. Similarly, the correctness of the implementation is verified by syntactical processes, as we shall elaborate upon in the sequel.

Let us examine more closely what is involved in implementing an abstract data type A on another one, C. The result will be a module representing objects of A in terms of those of C, and operations and predicates of A by means of procedures using operations and predicates of C.

We can abstract a little from the actual procedure texts by replacing them by specifications of their input-output behaviours. These amount

---

[1] More details on some of these points are planned to appear in the forthcoming reports Logical Specifications: 3. Extensions of Specifications and Logical Specifications: 5. Implementations of Specifications.

to (perhaps incomplete) definitions of the operations and predicates of A in terms of those of C and can be regarded as axioms involving both the symbols of A and of C. Similarly, the representation part describes the abstract sorts in terms of the concrete ones, which can be abstracted into axioms introducing the new sorts, and capturing (some of) the so-called representation invariants (Guttag 1977).

With this abstraction in mind we are ready to describe this situation in terms of formal specifications, i. e. theories presented by axioms (Turski and Maibaum 1987; Veloso 1987).

One extends the concrete specification C by adding symbols to correspond to the abstract ones in A, perhaps together with some auxiliary symbols. Since one does not wish to disturb the given concrete specification C, this extension[1] B should not impose any new constraints on C. This can be formulated by requiring the extension B of C to be conservative (Shoenfield 1967) (or non-creative) in the sense that B adds no new consequence to C in the language of the latter.

One then wishes to correlate the abstract symbols in A to corresponding ones in B, much as procedure calls are correlated to their corresponding bodies. But, the properties of A are important, for instance in guaranteeing the correctness of the abstract program supported by A. Thus, in translating from A to B, one wishes to preserve the properties of A as given by its axioms. Thus, one needs a translation $i: A \rightarrow B$ that is an interpretation[2] of theories (Shoenfield 1967) in the sense that it translates each consequence of A to a consequence of B.

We thus arrive at the concept of an implementation of A on C as an interpretation i of A into a conservative extension B (sometimes called a mediating specification) of C (Maibaum *et al.* 1985). This is depicted as an implementation 'triangle' in figure 1.1 and is often called a "canonical implementation step" (Turski and Maibaum 1987).

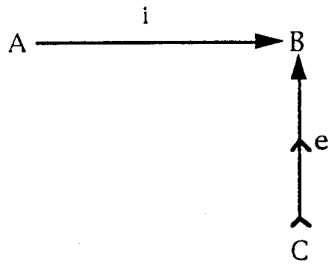

Fig. 1.1: Implementation 'triangle'

---

[1] Extensions are examined in more detail in in the forthcoming report Logical Specifications: 3. Extensions of Specifications.

[2] Interpretations are examined in more detail in in the forthcoming report Logical Specifications: 4. Interpretations of Specifications.

1.4 Composing implementations as theories

In stepwise development it is highly desirable to be able to compose refinement steps in a natural way. Let us consider the situation depicted in figure 1.2. Here, one has a first implementation of A on C (with mediating specification B) and a second implementation of C on E (with mediating specification D). Now, one would like to compose these two implementations, in an easy and natural manner, so as to obtain a composite implementation of A directly on E. An immediate question that arises is: what would its mediating specification be?

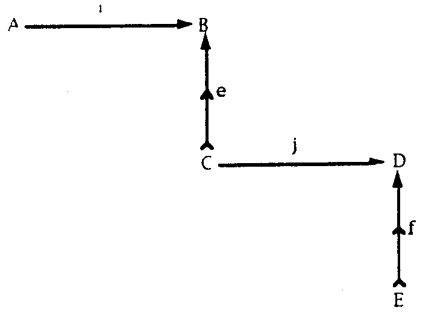

Fig. 1.2: Composition of implementation steps

This is where an important property, the so-called Modularisation Property [1], comes into play. It will allow one to obtain such a mediating specification M, together with the required interpretation k of B into M and a conservative extension g of E into M. In other words, it will enable one to complete the rectangle, thereby obtaining a composite implementation of A directly on D, consisting of a composite interpretation of A into M together with a composite conservative extension of D into M, as illustrated in figure 1.3.

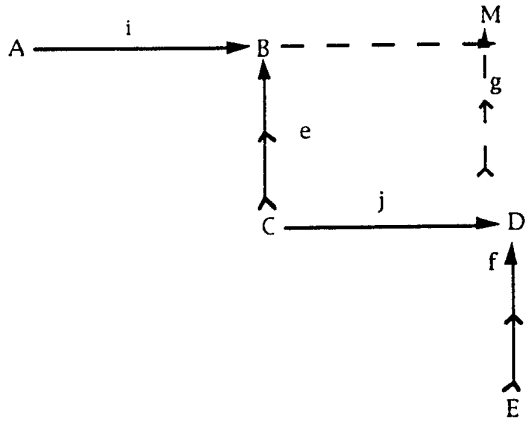

Fig. 1.3: Composite implementation step

[1] A more detailed discussion of the Modularisaton Property is planned to appear in the forthcoming report Logical Specifications: 4. Interpretations of Specifications.

Thus, an immediate benefit of this view is the iteration of implementation steps: an implementation of A by C "composes" naturally with one of C by E to yield an implementation of A by E. Here it is worthwhile noting that this composition mimics exactly what a programmer does in simply putting together the corresponding modules.

Another dividend stems from the fact that this view concentrates on the logical aspects of implementation. For, recall that in passing from C to B we add formulae rather than programs. These formulae record the design decisions taken in the implementation, not yet their actual coding into a program text. Therefore, we achieve orthogonality: the process of coding actual modules is independent of - and can proceed in parallel with - the process of further (logical) refinement, say, in implementing C by E. A simple example of this fact is "families of programs" (Parnas 1979). The successive refinements record the various design decisions, allowing the development of, say, sorting algorithms, naturally classified into families (Darlington 1978).

1.5 Parameterised specifications as theories[1]

One of the long standing research goals in work on formal specifications is the provision of standard building blocks from which larger specifications might be constructed and that may be re-used in different situations. In particular, the structuring of a specification into a "context" and "parameter" has been found to be particularly useful. The idea is that the context can be plugged into different situations by appropriate choice of values (instances) for the parameters. Such structured specifications are called parameterised (or generic) specifications (Ehrich 1982; Ehrig and Mahr 1985).

Let us consider a simple example: SEQ[DATA] (sequences of, as yet, unspecified values), where DATA is the formal parameter and SEQ is what we referred to above as the context. Thus, DATA should be a part of SEQ[DATA]. One may visualise this situation as in figure 1.4.

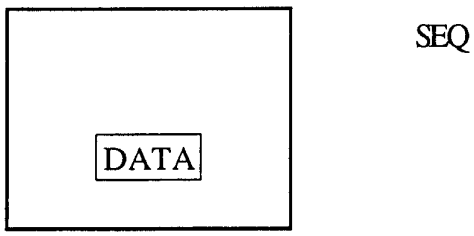

Fig. 1.4: Parameterised data type SEQ[DATA].

[1] More details on some of these points are planned to appear in the forthcoming reports Logical Specifications: 4. Interpretations of Specifications and Logical Specifications: 6. Parameterised Specifications.

Now, one would like to instantiate DATA by various actual arguments to get 'normal' specifications. So, the replacement of DATA by a specification NAT, of natural numbers, should give a specification for sequences of naturals; similarly, the replacement of DATA by INT should give a specification for sequences of integers. One wishes to instantiate DATA in SEQ[DATA] by NAT to obtain SEQ[NAT] by 'replacing' the formal parameter DATA by the actual argument NAT. Our intuition tells us that SEQ[NAT] should look like figure 1.5. In other words, one just replaces the DATA part by NAT within the context SEQ. Thus, for each given specification for DATA, SEQ[DATA] produces an instantiated version, like SEQ[NAT]. In an analogous manner one can instantiate SEQ[DATA] to SEQ[INT].

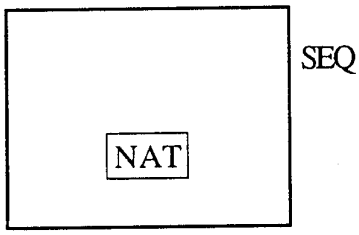

Fig.1.5: Instantiated data type SEQ[NAT].

The above intuition suggests regarding SEQ[DATA] as a function on specifications. Indeed, this is the idea underlying the semantics of a parameterised specification as a (perhaps partial) function from models to models, or as a (partial) function from specifications to specifications.

Another viewpoint is provided by concentrating on the properties of the specifications (Maibaum *et al.* 1985; Maibaum *et al.* 1991). First, one should expect SEQ[NAT] to inherit all the properties of SEQ[DATA] that concern only sequences, such as tail[cons(d,s)]=s. Also, DATA is supposed to be a part of SEQ[DATA], but not an arbitrary one, in the following sense. In going from DATA to SEQ[DATA], one would not expect to gain any more knowledge about DATA. In other words, no new constraints on DATA are placed by the addition of the context SEQ[ ]. (This is not to say that every parameter is appropriate for every context.)

The simple tools of conservative extension and interpretations between theories provide us with a quite straightforward account of parameterisation which is consistent the treatment in the algebraic setting. The parameterised types have specifications that are essentially the same as those of normal types. Thus SEQ[DATA] should be a specification just like SEQ[NAT]. Their meanings (theories) are the same as for normal specifications and not (partial) functions between specifications or models. Instantiation of a formal parameter by an

actual argument rests on a straightforward application of the Modularisation property[1].

The situation in instantiating a formal parameter to an actual argument is similar to the one encountered in composing implementations. The Modularisation Property completes the rectangle, thereby yielding the resulting instantiated specification. Here it is worthwhile noting that the construction of this instantiated specification mimics exactly what was suggested above in figures 1.4 and 1.5.

1.6 Liberal specifications and errors[2]

Another important benefit of viewing specifications as (presentations of) theories is not exaggerating the importance of errors. Asking for the value of "head of nil" is a semantical question. We can declare it to be an error; we can postpone the decision of what value to return to a later implementation stage (Maibaum *et al.* 1984); or, if the specific value assigned to this expression is not required to prove something about the system we are developing, we need not say anything about it at all!

The rationale behind liberal specification is not overspecifying. This gives more freedom in specifying abstractions and in implementing them after taking some design decisions. That is, incomplete knowledge is, not only bearable, but also often desirable. This is where a property-oriented, as opposed to model-oriented, approach is convenient: the properties of an abstraction are the consequences of its specification, which may very well be silent about some details. Some of these details may, or may not, be refined at subsequent development steps.

Summing up, the basic ideas of this approach are as follows. Programming deals with syntactical entities in an abstract manner. Properties of objects make up the stuff programming relies on. In other words, programming amounts to manipulation of presentations of theories.

1.7 General overview of the chapter

We have presented some advantages of viewing program development as manipulations of theories. In a nutshell, they amount to enabling clear distinctions amongst specification of desired behaviour, design decisions and program text throughout the whole process (perhaps iterated, as in stepwise refinements). In doing so, we sometimes resort to a terminology reminiscent of first-order logic. However, it should be

---

[1] More details on parameterisation and instantiation are planned to appear in the forthcoming reports Logical Specifications: 4. Interpretations of Specifications and Logical Specifications: 6. Parameterised Specifications.

[2] More details on some of these points are planned to appear in the forthcoming report Logical Specifications: 3. Extensions of Specifications.

clear that the spirit of this viewpoint can be carried over to the more general framework of a linguistic system (Turski and Maibaum 1987).

In the sequel, we will use mainly many-sorted first-order logic to illustrate how such a property-oriented approach to formal specifications can be realised in a conventional setting. We shall generally employ the usual terminology and notation for logical concepts, for which the reader is referred to standard textbooks, for instance (Enderton 1972; Ebbinghaus *et al.* 1984; Shoenfield 1967; van Dalen 1989). These will be briefly reviewed as they are needed.

This chapter is divided into five sections:

1. Introduction and Overview
2. Specifications as Presentations
3. Extensions of Specifications
4. Interpretation of Specifications
5. Implementation of Specifications
6. Parameterisation of Specifications
7. Conclusion: Retrospect and Prospect.

1.8 References

Arbib, M. and Mannes, E. (1975). *Arrows, Structures and Functors : the Categorical Imperative*. Academic Press, New York.

Barwise, J. ed. (1977). *Handbook of Mathematical Logic*. North-Holland, Amsterdam.

Bauer, F., L. and Wössner, H. (1982). *Algorithmic Language and Program Development*. Springer Verlag, Berlin.

Darlington, J. (1978). A synthesis of several sorting algorithms. *Acta Informatica*, **11** (1), 1-30.

Ebbinghaus, H. D., Flum, J. and Thomas, W. (1984). *Mathematical Logic*. Springer-Verlag, Berlin.

Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press; New York.

Ehrich, H.-D. (1982). On the theory of specification, implementation and parameterization of abstract data types. *J. ACM*, **29** (1), 206-227.

Ehrig, H. and Mahr, B. (1985) *Fundamentals of Algebraic Specifications, 1: Equations and Initial Semantics*. Springer-Verlag, Berlin.

Ledgard, H. and Taylor, R. W. (1977) Two views on data abstraction. *Comm. Assoc. Comput. Mach.*, **20** (6), 382-384.

Guttag, J. V (1977). Abstract data types and the development of data structures. *Comm. Assoc. Comput. Mach.*, **20** (6), 396-404.

Maibaum, T. S. E. (1986). The role of abstraction in program development. In Kugler, H.-J. ed. *Information Processing '86*. North-Holland, Amsterdam, 135-142.

Maibaum, T. S. E., Sadler, M. R. and Veloso, P. A. S. (1984). Logical specification and implementation. In Joseph, M. and Shyamasundar R. eds. *Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, Berlin, 13-30.

Maibaum, T. S. E. and Turski, W. M. (1984). On what exactly is going on when software is developed step-by-step. *tProc. 7h Intern. Conf. on Software Engin.* IEEE Computer Society, Los Angeles, 528-533.

Maibaum, T. S. E, Veloso, P. A. S. and Sadler, M. R. (1985). A theory of abstract data types for program development: bridging the gap?. In Ehrig, H., Floyd, C., Nivat, M. and Thatcher, J. eds. *Formal Methods and Software Development; vol. 2: Colloquium on Software Engineering*. Springer-Verlag, Berlin, 214-230.

Maibaum, T. S. E, Veloso, P. A. S. and Sadler, M. R. (1991). A logical approach to specification and implementation of abstract data types. Imperial College of Science, Technology and Medicine, Dept. of Computing Res. Rept. DoC 91/47, London.

Manna, Z. (1974). *The Mathematical Theory of Computation*. McGraw-Hill, New York.

Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Trans. Software Engin.*, **5** (2), 128-138.

Shoenfield, J. R. (1967). *Mathematical Logic*. Addison-Wesley, Reading.

Smith, D. R. (1985).The Design of Divide and Conquer Algorithms. *Science Computer Programming*, **5** 37-58.

Smith, D. R. (1990). Algorithm theories and design tactics". *Science of Computer Programming.*, **14**, 305-321.

Smith, D. R. (1992). Constructing specification morphisms. Kestrel Institute, Tech. Rept. KES.U.92.1, Palo Alto.

Turski, W. M and Maibaum, T. S. E. (1987). *The Specification of Computer Programs*. Addison-Wesley, Wokingham, .

van Dalen, D. (1989). *Logic and Structure* (2nd edn, 3rd prt). Springer-Verlag, Berlin.

Veloso, P. A. S. (1984). Outlines of a mathematical theory of general problems. *Philosophia Naturalis*, **21** (2/4), 354-362.

Veloso, P. A. S. (1985). On abstraction in programming and problem solving. *2nd Intern. Conf. on Systems Research, Informatics and Cybernetics*. Baden-Baden.

Veloso, P. A. S. (1987). *Verificação e Estruturação de Programas com Tipos de Dados.* Edgard Blücher, São Paulo.

Veloso, P. A. S. (1987). On the concepts of problem and problem-solving method. *Decision Support Systems*, **3** (2), 133-139.

Veloso, P. A. S. (1988). Problem solving by interpretation of theories. In Carnielli, W. A. ; Alcântara, L. P. eds. *Methods and Applications of Mathematical Logic.* American Mathematical Society, Providence, . 241-250.

Veloso, P. A. S. (1992). On the modularisation theorem for logical specifications: its role and proof. PUC - RJ, Dept. Informática Res. Rept. MCC 17/92, Rio de Janeiro.

Veloso, P. A. S., Maibaum, T. S. E. and Sadler, M. R. (1985). Program development and theory manipulation. In *Proc. 3rd Intern. Workshop on Software Specification and Design.* IEEE Computer Society, Los Angeles, 228-232.

Veloso, P. A. S. and Maibaum, T. S. E. (1992). On the Modularisation Theorem for logical specifications. Imperial College of Science, Technology & Medicine, Dept. of Computing Res. Rept. DoC 92/35, London.

Veloso, P. A. S. and Veloso. S. R. M. (1981). Problem decomposition and reduction: applicability, soundness, completeness. In Trappl, R.; Klir, J. ; Pichler, F. eds. *Progress in Cybernetics and Systems Research.* Hemisphere, Washington, DC, 199-203.