Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

61

The Vienna Development Method: The Meta-Language

Edited by
D. Bjørner and C. B. Jones

Springer-Verlag
Berlin Heidelberg New York 1978

Editors
Dines Bjørner
Department of Computer Science
Building 343 and 344
Technical University of Denmark
DK-2800 Lyngby

Cliff B. Jones
IBM International Education Centre
Chaussee de Bruxelles 135
B-1310 La Hulpe

Library of Congress Cataloging in Publication Data
Main entry under title:

The Vienna development method.

(Lecture notes in computer science ; 61)
Bibliography: p.
Includes index.
1. Programming languages (Electronic computers)--
Addresses, essays, lectures. I. Bjørner, Dines.
II. Jones, Cliff B., 1944- III. Title: Meta-
language. IV. Series.
QA76.7.V53 001.6'424 78-7232

AMS Subject Classifications (1970): 68-02, 68A05, 68A30
CR Subject Classifications (1974):

ISBN 3-540-08766-4 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-08766-4 Springer-Verlag New York Heidelberg Berlin

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-5432

CONTENTS

All papers lists their CONTENTS at their very beginning.

INTRODUCTION

The purpose of this volume is to provide a summary of a body of work which has reached a relatively stable state. The work is, however, far from complete (in the sense - even - that the authors whose work is presented here feel that they have satisfactory solutions to the problems they set out to solve.) Notwithstanding their own recognition of difficulties and shortcomings in the current presentation, the authors hope that what has been achieved may be of use to others. Furthermore, a summary of the results of any significant effort may be hoped to stimulate the work of researchworkers.

The purpose of this introduction is, of course, to introduce the volume. Firstly an outline is provided of the so-called "Vienna Development Method" and some motivation offered of the meta-language which is the part of the method of concern in this volume. Following this a review is provided of other work done in the Vienna Laboratory, in order to put what is presented here in context.

Vienna Development Method

Before entering into the description itself, it is perhaps worth spending a few moments on the name itself in the hope to avoid a misunderstanding which could arise. The earlier work of the Vienna laboratory developed a meta-language for definitions which became known as the "Vienna Definition Language" (*VDL*). The relationship between the work presented in this volume and the *VDL* is discussed below. Of course, the earlier meta-language had a large influence on the later one. But even here there is a sharp difference between the two (technically, *VDL* was designed for writing operational definitions whilst the meta-language used in *VDM* is intended for the presentation of denotational definitions). Moreover, as the name suggests, the Vienna Development Method (abbreviated to *VDM* in this volume) is much more than just a meta-language.

Turning now to the content of *VDM*. The "method" is meant to be a systematic approach to the development of large computer software systems. Strictly, there is nothing which restricts the application to software but no attempts have been made to use the approach on hardware design. The key proposals are simple and by no means unique to *VDM*. As indicated in *fig. 1*, the first objective in getting any complex system

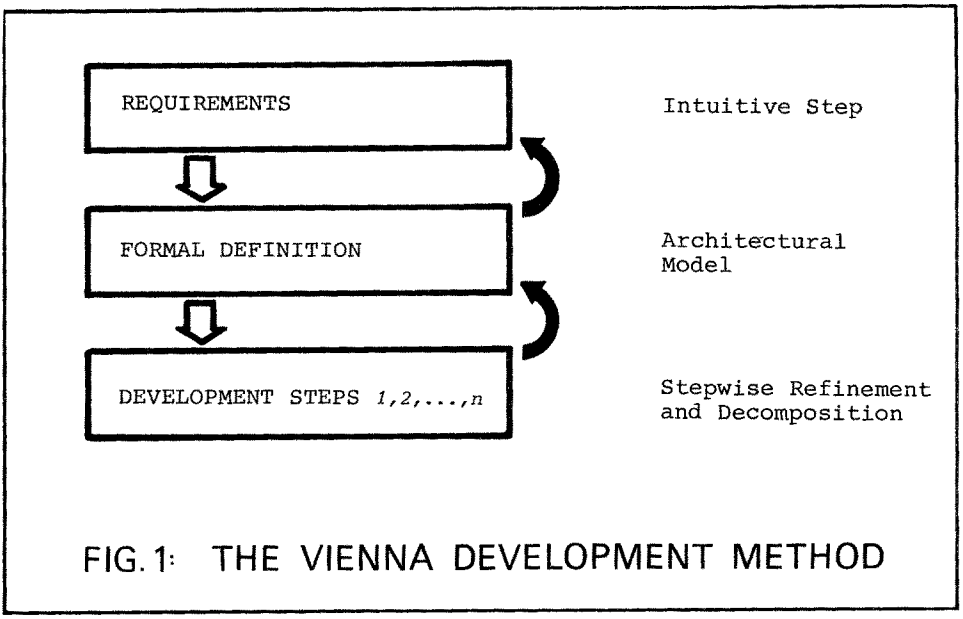

FIG. 1: THE VIENNA DEVELOPMENT METHOD

under control is the construction of a formal definition of the required function. This specification is a reference point for the subsequent development. For a large system there will follow a sequence of development steps. The downward pointing arrows are suggestive of a time sequence. Because it is often necessary to 'think ahead' in a design process, it is more accurate to view the arrows as showing the relationships between the final stages of the documentation. Thus, a reader may understand the design of a system in a strictly top-down way although this is an idealization of the actual (iterative) design process.

The backward links in *fig. 1* relate to another key point of *VDM*. At each stage of development a justification is provided. Just as the overall process begins with a specification, each intermediate development step can be viewed as presenting a solution to a specification (all that distinguishes the last 'implementation' phase is that instead of generating new specifications, all of the required units are available). The aim of a justification is to document why the proposed solution is believed to fulfil its specification. Such justifications can be presented at different levels of formality. But without entering here into a discussion of what level might be chosen in various cir-

cumstances, it should be clear that justifications documented during the development process are likely to be both much more intuitive and of more use in early detection of design errors than any attempt to construct a proof for a complete system after construction.

Decomposition and correctness arguments are two key points: a third is the use of abstraction to handle complexity. In the papers in this volume it will be made clear how a specification can be viewed as an abstract model of the system to be specified. Such an abstract model will make extensive use of abstract data objects (see Bjørner 78b): it is the choice of appropriate abstractions which can make a short and readable formal specification. The use of abstract objects during the development process and their refinement to objects which are representable in the eventual realization is discussed in Bjørner 77a, Jones 77a.

Whilst *VDM* was first envisaged for languages and the development of their processors, the definition methods have subsequently been applied to other "systems" - see Hansal 76, Nilsson 76, Madsen 77.

The Meta-Language

Having employed an internal name ("Vienna Development Method") for the first part of the title of this volume, the anonymous sub-title may cause some surprise. In fact there is an internal name ("Meta-IV") for the meta-language used in *VDM* so its omission can be guessed to have strong grounds. The point is that in defining and developing systems a number of concepts have been used; it has of course been necessary to use a notation to manipulate the concepts. It is, however, the concepts which are important not the particular concrete syntax used for expressions realizing these concepts. It was felt by the authors that making wide use of a name for the meta-language might focus attention on the wrong issues and so it has been avoided. Furthermore, it should be made clear that the meta-language is not "closed" in that (well defined) extensions can be made without fear of a 'standards committee'.

The need for a meta-language should be clear. Decomposition has been shown to be a part of *VDM* and this only makes sence if something is written about each stage of development. If something is to be written it must use some language. In view of the systematic approach being taken

to development and, in particular, the use of justifications, it should be obvious that a formal notation is required. Given that abstract objects are required, choosing established notation (e.g. for sets) would appear to be wise. Surprisingly, the decision to use 'standard' mathematical notation for standard things has actually caused some people to object! The clue is a concern, which is unfounded, that the use of notation from a branch of mathematics implies that a deep knowledge thereof is required.

Another influence on the meta-language has been programming languages. Just as choosing understood mathematical objects can aid the reader, the use of sequencing constructs from programming languages (e.g. *if then else*) can enhance the readability of a definition. Of course, all such constructs employed must themselves be precisely defined - this is tackled in Jones 78a.

The major inpetus towards denotational semantics has come from Oxford University. There is, however, a striking difference in the appearance of definitions created by Oxford or Vienna. This subject is returned to later in the volume but it is important to remember the objectives of the different groups in order to avoid (erroneously) seeking a "correct" choice. The Oxford group have been interested in the foundations of their meta-language and its use on examples small enough to facilitate complete proofs; the Vienna group was forced to take a more engineering approach when faced with languages like PL/I.

Several things have been deliberately excluded from this volume. Firstly the subject of concrete syntax is well-documented elsewhere (e.g. Uzgalis 77) and thus, whilst an important part of a system definition, it is ignored. Secondly, there is no formal description of methods for defining parallelism. Thirdly the relationship to other methods is only discussed by P. Lucas: this is not a comment on the other authors' views of, for example, axiomatic definitions.

Relation to Earlier Vienna Work

The purpose of this section is to identify the main differences between the work presented in this volume and that done earlier in the Vienna Laboratory. For this objective a complete historical view would be unnecessarily long and is not attempted. (This section fits, logically, into the introduction, but the reader who is unfamiliar with the material will find it more beneficial to first read both the more general historical review - Lucas 78 - and one of the descriptions of the current meta-language - Bjørner 78b or Jones 78a.)

In order to have precise names for the two phases that are to be compared, "*VDL*" and "*VDM*" will be used. "Vienna Definition Language" (*VDL*) was a term coined and used most widely in North America and identifies the language definition notation developed and used in the Vienna Laboratory during the 1960's. The Vienna Development Method (*VDM*) has been described above. It owes much to the earlier work but differs in some inportant technical respects: these differences and their motivations are the subject here.

We begin by briefly reviewing some of the important documents relating to the earlier work. The Vienna group had constructed a compiler for ALGOL 60 and following this work was asked to undertake a formal definition of the PL/I language. The acknowledged basis of this work was the papers McCarthy 63a, Landin 64, Landin 65, Elgot 64 and the Baden TC2 conference of 1964 (see Steel 66 - especially McCarthy 66). The first "tentative steps" towards a style for "Universal Language Description" (*ULD*) are recorded in Bandat 65. It is worth quoting the objective as described by P. Lucas:

> "The result may serve as a vehicle for language design groups, implementation groups, and as a useful background for educated and sophisticated programmers. ... It should be possible to formulate and prove statements about the object language."

The first version of the PL/I definition did not cover the complete language when printed in 1966. The 1968 version 2 was essentially complete and became an internal control and reference tool for the evolving language. This evolution led to the requirement for a third version and subsequent (extensive) revisions. It is interesting to review the structure of this third version:

Compile Time Facilities	Fleck 69
Concrete Syntax	Urschler 69a
Translation Concrete to Abstract	Urschler 69b
Abstract Syntax and Interpreter	Walk 69
Informal Introduction	Alber 69

The *VDL* notation is a means of describing abstract interpreters (see Lucas 78 for definition of this approach.) A very general view was taken of objects and their manipulation (cf. the "μ" function) and the control component was made part of the state of the interpreter. This meant that extremely "flexible" interpreters could be constructed which may have been one of the reasons why *VDL* was quickly adopted by a number of groups both in and outside IBM. To give just a few references: Lauer 68, Zimmermann 69, Lee 72, Moser 70a (this is one of several attempts to use the meta-language for the descriptions of "systems").

The best overview of the *VDL* work is probably still Lucas 69 along with Bekic 70b on the subject of storage models. Other reviews are available in Wegner 72 and Ollengren 75. (Although this section is not a full historical survey it would be remiss not to mention the guidance provided by H. Zemanek - see, for example, Zemanek 66).

Rather than listing the users of *VDL* it will be more germane to consider how such definitions were used in the justification of implementations of defined languages. Just as with the work on program development, the initial work concentrated on proofs: once this basis was laid, attention was turned to systematic development methods. Realizing his earlier (quoted) hopes P. Lucas was able to demonstrate the use of a *VDL* definition in proving an implementation correct in Lucas 68. A considerable amount of work in this direction was then undertaken and is reviewed in Jones 71. Much of this work was made more difficult than one felt was necessary by the "flexibility" of the abstract interpreters which could be written using *VDL*. In particular the ability to explicitly change the control meant that inductive proofs over the structure of (abstract) programs were not, in general, valid; and the inclusion of objects like the environment in a "Grand State" complicated proofs. Although the proofs in Jones 71 avoided the former problem, the latter led to some of the longest proofs in the paper. The attempts to use *VDL* definitions as a basis for systematic development of implementations was, then, providing indications that a change of definition style might be worthwhile.

Other important questions were being raised on the style of definition. Of great importance were the observations in Bekić 70a that a more "Mathematical" style could avoid much unnecessary detail. One of the problems which had caused the use of the control in *VDL* was providing a model for goto statements. An alternative "exit approach" had been described in Henhapl 70a. A "functional" definition of ALGOL 60 used this approach in Allen 72. Furthermore, this definition had used a "small state" in that the environment was made a separate argument to the defining functions. This definition is not, however, "mathematical" ("denotational") and is unnecessarily complicated by the avoidance of combinators for frequently recurring patterns. (Other work on the question of "style" had, of course, been pursued - see, for example, Lauer 71, Hoare 69, Hoare 74).

The "denotational" approach is characterized in Lucas 78. The work of the Oxford group is well documented in Scott 71, Mosses 74, Stoy 74 (the most readable account) and Milne 76.

Towards the end of 1972 the Vienna group again turned their attention to the problem of systematically developing a compiler from a language definition. The overall approach adopted has been termed the "Vienna Development Method". Based on the above comments it should be no surprise that a "denotational" approach was adopted for the definition itself. (Using, however, the exit approach rather than "continuations" - see Jones 78b). This change was, in fact, less drastic than some authors choose to suggest. It is possible to read a denotational definition as an abstract interpreter. However, there is a denotational "rule" which requires that the denotations of compound objects should depend only on the denotations of their components: this rule leads one to the construction of definitions which are much clearer and easier to reason about. In fact the change from operational to denotational style was further masked by a preservation of an overall Vienna "flavour". This flavour comes from the choice of appropriate abstractions for source and semantic objects and a writing style which aims at readability rather than conciseness.

The meta-language actually adopted ("Meta-IV") is used to define major portions of PL/I (as given in ECMA 74 - interestingly a "formal" standards document written as an abstract interpreter) in Bekić 74. The project went on to consider how this definition would be used to construct a compiler. An indication of the interface problem which results from

using a typical "product oriented" front-end, is shown in Weissenböck 75. A concern is often expressed as to how one can check that a formal definition captures one's intuitive notion of a language. Since the latter is inherently informal, the short answer is that one can not so do. But certain consistency conditions can be established and this is the subject of Izbicki 75. Although the project was not pursued to the stage of a running compiler, the overall method used is described in Jones 76a.

Again the meta-language developed for the *VDM* phase has been used by a number of other groups. The only significant IBM publication is Hansal 76 which is interesting because it addresses the problem of relational data bases. Externally, Nilsson 76 and many working documents of the Technical University of Denmark have used "Meta-IV" (see also Bjørner 77b).

There are still a number of open issues. The problem of defining arbitrary merging and/or parallelism was tackled in Bekic 74 but the technique used has not yet been defined in a satisfactorily "Mathematical" way - see Bekic 71 and Milner 73. Another area of future research is outlined in Mosses 77. In the general view of making definitions more abstract, the rôle of the environment is questioned in Jones 70.

On the Definition of PL/I

Bekić 74 contains a definition of most of the non-I/O parts of PL/I as described in ECMA 74. (Some of the input-output statements were defined by W. Pachl, but the work is not published). The ALGOL 60 definition in this volume (Henhapl 78) has benefited from that work and exhibits many of the formulations used earlier. There are, unfortunately, minor notational differences to be faced in reading the earlier work. The significant differences, however, derive from the extra "richness" of PL/I and the main aspects of the relevant formulations are reviewed here. (Again, this section should be skipped at first reading).

As mentioned above, some attempt was made in Bekić 74 to cover the problem of arbitrary order. In particular the order of access to variables anywhere within expressions is not constrained. The "," combinator was introduced to tackle this problem.

Because PL/I offers a larger set of ways of building aggregates than is available in ALGOL 60, a more implicit model of storage is used. Furthermore, the normal way of governing the lifetime of variables in ALGOL 60 becomes one of four ways available in PL/I: this normal way is called *AUTOMATIC*; the "own" variables of ALGOL 60 correspond roughly to *STATIC*; in addition PL/I offers *BASED* and *DEFINED* storage classes. With *BASED* variables explicit allocation statements must be executed and there is no automatic freeing. There are a number of expressions involved such as references to pointer variables and the main interest is in showing when, in what environment and with what exception conditions (see below) these various expressions are evaluated.

As well as parameters of type *ENTRY* (i.e. procedure) and *LABEL*, PL/I permits variables of these types. Furthermore, their use is not statically constrained, as is the case in ALGOL 68, to prevent attempted access to entities local to a block after the lifetime of that block. Therefore, the problem of checking for "past activations" had to be tackled in the PL/I definition.

Perhaps the most interesting extension is the use of "ON conditions" and "condition built-in-functions/pseudo-variables". ON statements ca be modelled as assignments to *ENTRY* variables (notice they are dynamically, not lexicographically, inherited). The effect of encountering a condition (whether *SIGNAL*led or implicitly raised) can be modelled by a call of the procedure which is currently assigned to the appropriate variable. The pseudo-variables used for investigating and returning values from these procedures behave like global variables.

For further details on the PL/I model, the reader is referred to the "annotation" section of Bekić 74.

The Structure of this Volume

The papers of this volume can be grouped in four categories.

(I) The first paper:

On the Formalization of Programming Languages: Early History & Main Approaches

by Peter Lucas sets the stage. It discusses the main approaches to language definition, the intrinsic aspects of the problem area and their origins. As such the paper provides a frame for the remainder of the volume.

(II) The next two papers:

Programming in the Meta-Language: A Tutorial

by Dines Bjørner, and:

The Meta-Language: A Reference Manual

by Cliff B. Jones give complementary descriptions of the meta-language. The tutorial is a partly informal introduction to, partly comprehensive primer for, the meta-language. The reference manual gives precise semantics definitions of the more important meta-language constructs. The tutorial is primarily aimed at persons new to formal definitions, but with some background in (ALGOL-like) programming. The reference manual, in contrast, is primarily aimed at people, familiar with the basic ideas of denotational semantics, who wishes to understand the meta-language. Comprehension of the tutorial is otherwise a sufficient prerequisite for any other paper of this volume. The tutorial describes constructs not formally covered by the reference manual. Any such construct can, however, be simply reduced to simple combinations of constructs formally covered by the reference manual. It is in this sense we say that the language described in the tutorial is 'larger' than that defined by the reference manual.

(III) The fourth paper:

Denotational Semantics of Goto: An Exit Formulation and its Relation to Continuations

by Cliff B. Jones, brings focus on a major factor distinguishing the Oxford, Scott-Strachey, School of expressing Denotational Semantics, from the current Vienna School. As such the paper contributes to a deeper understanding of the *VDM* meta-language by analyzing one of its combinators, the *exit* construct.

(IV) The last group of papers exhibits actual abstractions:

A Formal Definition of ALGOL 60 as Described in the 1975 Modified Report

by Wolfgang Henhapl & Cliff B. Jones presents the latest in a number of ALGOL 60 definitions. Over its only 32 pages of text and formulae it demonstrates, we believe, the power of the abstractional techniques used, and the meta-language tool described earlier, by giving a very readable and neat denotational semantics definition. The last paper of the volume:

Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition.

by Dines Bjørner summarizes a number of complementing & contrasting abstract modeling techniques. The first example is chosen to indicate the applicability of the software abstraction ideas to other than conventional programming languages; the second in order to illustrate various state modeling techniques & also the unusual On-Condition Language construct.

The volume ends with a unified bibliography recording the literature referred to in the various papers.

ACKNOWLEDGEMENTS

The editors of this volume gratefully acknowledge the Computer Science Department of The Technical University of Denmark and the European Systems Research Institute of the IBM Corporation for their kind support, enabling us to prepare this volume.

The editors are especially happy to thank the co-authors: Wolfgang Hanhapl & Peter Lucas, for their much appreciated contributions.

These latter actually stretches back over the many years the authors were members of the IBM Vienna Laboratory. To all of our colleagues, some of them still there, goes our most sincerely felt appreciation for the seemingly never-ending source of inspiration they represent.

Very special, deep and fond thanks goes to Prof. Heinz Zemanek for having created such unique working environments; and to Dr. Hans Bekić for his unwavering high standards which kept us straight.

Finally all co-authors join the editors in expressing their indebtness for the expert and untiring assistance of Mrs. Annie Rasmussen and Mrs. Jytte Søllested.

Dines Bjørner & Cliff B. Jones

Montpellier, France

January, 1978