Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

428

D. Bjørner C.A.R. Hoare
H. Langmaack (Eds.)

VDM '90
VDM and Z – Formal Methods in Software Development

Third International Symposium of VDM Europe
Kiel, FRG, April 17–21, 1990
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo Hong Kong

Editors

D. Bjørner
Department of Computer Science, Technical University of Denmark
Building 344–345, DK-2800 Lyngby, Denmark

C.A.R. Hoare
Programming Research Group, Oxford University
8–11 Keble Road, GB-Oxford OX1 3QD, England

H. Langmaack
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preußerstraße 1–9, D-2300 Kiel 1, FRG

CR Subject Classification (1987): F.3.1, D.2.1, D.2.4

ISBN 3-540-52513-0 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-52513-0 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210 – Printed on acid-free paper

Foreword

The present volume is the third in a series of VDM Symposia Proceedings. The previous two VDM Symposia were held in March 1987 in Brussels, Belgium, and in September 1988 in Dublin, Ireland. Their proceedings were published by Springer–Verlag in the same series as this volume, as Lecture Notes in Computer Science volumes 252 ([1]) and 328 ([2]), respectively.

VDM and Z are acronyms. VDM stands for Vienna Development Method, while Z refers to Zermelo, a fine mathematician whose name is associated with set theory. VDM arose out of an industrial project at the IBM Vienna Laboratory in the early 1970s. Z was first masterminded by J.-R. Abrial, who laid down the basic ideas while at Oxford in the very early 1980s. VDM later moved into academia, lacking industrial support at first, while Z had the good fortune of getting industrial support.

The preface by Tony Hoare gives an interesting view of the uniting and distinct features of VDM and Z.

The Commission of the European Communities (CEC) has had the fortunate vision to form, around 1985, VDM Europe, which is a group of industrial and academic software engineers, programmers and scientists interested in model-theoretic formal software development methods. VDM Europe advises the CEC. VDM Europe is partially supported by the CEC and meets in Brussels 3–4 times yearly to discuss practical, technical and scientific matters. These are: industrial experience and applications, tools, college and university education and industry training, specification and design methodology and techniques, formalisation and formal foundations, and standardisation. The British Standards Institute (BSI) is currently formulating a standard for the VDM Specification Language (SL) — a standard that will also be proposed to the ISO. That standard will contain fully formal, mathematical definitions of the semantics of the BSI VDM SL.

In the UK there is a group similar to VDM Europe called the Z Users Group.

While planning this symposium VDM Europe felt it reasonable, given the commonalities outlined in Tony Hoare's preface, to invite the Z community to join that of VDM in presenting our achievements.

Many computing science, programming and software engineering proceedings are published regularly. The ones by VDM Europe have the distinguishing mark, we believe, that they are concerned with bringing real theory to apply to real programming. This is true both for VDM and Z. In the proceedings of the VDM Symposia you will therefore find a refreshing blend of papers from industry and from academia. In fact you will witness an enjoyable cross fertilisation and cooperation between the two sides.

In Europe we are very much interested in methodology, semantics and techniques, that is, in understanding how we build and what it is that we are building.

Thus, non-European readers will find a somewhat biased selection of papers in these proceedings.

The papers in these proceedings basically fall into four major groups: Applications, Methodology, Formalisations, and Foundations. The Methodology group has been further subdivided into five areas: Specification Methodology, Design Methodology, Modularity, Object Orientedness, and Processes, Concurrency and Distributed Systems. Obviously the borderlines between these are fuzzy. We have chosen to present Applications first, and Foundations last. We wish the casual reader from industry to adopt the message of VDM and Z, namely apply formal, model-theoretic techniques in order to have fun, to achieve correct software, etc. Therefore we open with what is closest to the needs of industry by showing that indeed real applications can be formalised, and to great advantage. This is in contrast to an academic style of presentation which, after the fact, would normally present foundations first!

By a method we mean a set of procedures, used by humans, in selecting and applying techniques and tools in order to efficiently achieve the construction of efficient artifacts — here programs and software. By a formal method we mean one which permits mechanical support of rigorous development, that is, rigorous reasoning aided by mechanical type checking, semantic analysis, specification to design transformation, animation, proof assistance, verification and even theorem proving. Our sub-division into areas reflects techniques, albeit overlapping ones.

The Formalisation group has been sub-divided into Formalisation (!) and Type Systems. By formalisation we mean explaining formal systems, such as VDM and Z, in terms of some (other) formal specification system, including mathematics. By foundation we mean (examination of) the means in terms of which we explain. Type systems deal with a properly contained area of formalisation (and its 'mechanisation').

The invited papers were selected to reflect openness towards ideas that may not necessarily have arisen in the context of model-theoretic software development. Previous VDM Symposia featured presentations by Don Good: *Computing is a Physical Science*, and J.-R. Abrial: *The B Tool*. We are very fortunate, at this Symposium, to have three similarly exciting presentations: Ole-Johan Dahl: *Object Orientation and Formal Techniques*, Joseph A. Goguen: *Algebraic Approaches to Refinement* and Reiji Nakajima: *Modal Logic Programming*. We thank these invited speakers for their kind willingness to serve.

Kiel, FRG
February 1990

Dines Bjørner
Hans Langmaack

References

[1] *VDM'87: VDM — A Formal Method at Work* (Eds. D. Bjørner, C. B. Jones, M. Mac an Airchinnigh and E. J. Neuhold) Proceedings, VDM-Europe Symposium 1987, Brussels, Belgium, March 1987, Springer–Verlag, Lecture Notes in Computer Science, vol. 252, IX + 422 pages, 1987.

[2] *VDM'88: VDM — The Way Ahead* (Eds. R. Bloomfield, L. Marshall and R. Jones) Proceedings, 2nd VDM-Europe Symposium 1988, Dublin, Ireland, September 1988, Springer–Verlag, Lecture Notes in Computer Science, vol. 328, IX + 499 pages, 1988.

Preface

C.A.R. Hoare

I am delighted to be asked to welcome you to this international conference on VDM and Z. Your programme committee and organisers have worked hard to ensure that you will enjoy the occasion and benefit from it by increased understanding and confidence in the use of new methods in software engineering. My particular delight is that this conference, by its title and its content, is a welcome rapprochement between two technically sound and practically successful mathematical approaches to the specification and design of software.

The reason for their success is the same. Both VDM and Z insist that the specification of requirements should be formulated from the beginning at the highest possible level of abstraction, using all the available power of mathematics to describe the desirable, observable and testable properties of the product which is to be implemented. An abstract specification leaves greatest scope for exercise of ingenuity and judgement to meet targets of cost and efficiency in the subsequent design and implementation of the product; and correctness can be assured by observance of certain engineering procedures, which are soundly based on mathematical calculations.

VDM and Z share a common philosophy and a common method; but to many sympathetic supporters and potential users it seems both sad and strange that they seem to have different styles and notations, and to have given rise to disunity and disagreement among two apparently distinct schools of thought. The less sympathetic observer might be reminded of theological disputations of the Middle Ages leading to schisms, inquisitions and wars. How did such a situation arise? Was it due to historical, political or commercial accident? Or even to ignorance, incompetence or personal pride? Since I have been an actor in the history from the early days, I must acknowledge my share of the blame, if blame there be.

VDM, as its name implies, was originally designed as a method for sound development of concrete data structures from abstract specifications. It was based on proof of a commuting property like that of homomorphisms in algebra or natural transformations in category theory. Its designers were familiar with the important role of functions in mathematics, and the charming way in which they can be defined by recursion. They had an early and extensive experience of the use of formal methods in the specification, design and implementations of programming languages. These factors had an influence on the style and content of early versions of VDM.

The designers of Z started with the confidence provided by VDM that highly abstract specifications can be reliably implemented as programs. The design team were therefore able to concentrate attention on the earlier phases of requirements

capture and system-wide specification. J.-R. Abrial had a prior interest in the axiomatic foundations of mathematics, including the theory of sets and relations. At Oxford, we were also familiar with Dijkstra's predicate-based calculus of procedural programming, including non-determinism. This explains the slightly different emphasis placed on different branches of mathematics by VDM and Z.

Finally, at Oxford we were fortunate in obtaining an industrial sponsor interested primarily in improving the specification phase of the design of a large software product, and willing to postpone consideration of formal design and implementation. But the advantage of building on the success of VDM has also a sharp edge; the only point of the research was to suggest improvements and advances on what was already known and practised. And that was indeed a significant challenge, because the ultimate goal is that the designers and users of VDM would adapt and adopt the more successful technical and stylistic improvements suggested by our research. This has already happened: the mutual beneficial influence of Z and VDM is now so great that many of my remarks are of purely historical significance.

One of the first discoveries of the research by the Z team was the necessity of separating small chunks of formal material by paragraphs of informal prose, explaining the relationship between the formal symbols and reality, and motivating the decisions that are captured by the formalisation. The drafting of the informal prose was even more difficult to teach, learn and practice than the mastery of mathematical notations and concepts. But the need to simplify interleaving of formal and informal material led inevitably to the Z schema, with its free variables declared as its signature, and even to the box convention introduced to separate formula from text.

The next requirement is for modularity: it must be possible to formalise individual requirements separately, and join them together by simple connectives with obvious meanings. The simplest, most obvious and most necessary connective is conjunction: we want the product to meet this requirement *and* that requirement. We also need the kind of abstraction that can be achieved by disjunction: we want the product to have this property *or* that one, but the decision can be postponed perhaps even to the implementation. And a third propositional operator, negation, is useful in placing overall constraints on the behaviour of the product, for example to achieve safety. The schema calculus of Z is based on these simple propositional connectives.

Finally, we need to start our specifications with a simple description of the general structure of the system, with as few variables, parameters and operations as possible; and then leave this original text unchanged while later adding more detail. This can be achieved by embedding the initial general schema inside its particular extensions, thereby achieving a hierarchy like that exploited in object oriented programming.

That is my explanation of the early distinctive features of the Z schema calculus. Each one of them was subjected at the time to rigorous discussion and

evaluation, and only the most essential have survived. All of them arise almost inevitably from the need to simplify and clarify specifications. However, they are completely irrelevant in the design and development phase of system implementation, for which the notations, concepts and methods of VDM are designedly more appropriate. So the obvious advice to the practitioner is to use each notation for its primary intended purpose: Z for specification and VDM for development. Of course the transition between these two phases involves a translation, in which the separate schemas of the Z specification are assembled, and the VDM preconditions are calculated and checked. But such calculations are in any case essential to check the completeness and consistency of the specification; they are not just due to fortuitous notational incompatibility. Similar checks and changes of style and notation abound in all mature engineering disciplines. No one now specifies a computer architecture using only boolean algebra, but no computer can be designed without it.

So why was this obvious reconciliation of the complementary merits of Z and VDM not recognised earlier and more widely? The reason is equally obvious - it is scientifically inelegant and in practice very wasteful to use two notations if there is any conceivable chance that a single notation will do the whole job. So intensive research is required to see if either of the two notations can be used, extended or adapted for both purposes. And it is entirely reasonable to engage more than one team of scientists on this important task, and that each team should choose a different starting point, and that they should work in a spirit of friendly rivalry. But as scientists, everyone must have a deep respect for the merits and achievements of the rival team, and strong desire to learn from them. Preservation of scientific objectivity must be an overriding concern. The same goal is pursued by all, namely, the advancement of knowledge to a point at which we can give obvious and definitive answers to the basic questions.

1. Is one notation (or the other, or both) adequate for both specification and development?
2. If distinct notations are desirable, can this be justified by appeal to general scientific, mathematical or engineering principles?
3. In the latter case, what measures are recommended for easy and reliable transition between notations?

There are many software engineers and their managers who believe or hope that a single notation should be used not only for specification and development but also for prototyping and even production programming. Many millions of dollars have been expended in the search for such a notation and its implementation. But experience of other branches of mathematics and engineering suggests strongly that no uniform notation can be adequate for all purposes. And in the case of programming, we know both in theory and in practice that abstract notations appropriate

for specification are either incomputable or would take super-astronomical time to execute, even as prototypes.

In the light of recent excellent research, by teams concentrating on each of the two notations, I am now inclined to believe that there are certain fundamental differences between the structuring principles appropriate to specification and to development; and that we should begin to turn our attention to collaboration, in seeking the answers to question 2 and 3, which will ease the transition between these two essential phases of software engineering.

That is why this conference on VDM and Z is so welcome and so timely. It provides a forum at which we can exchange our understandings and experiences as an essential prelude to faster scientific progress. It also sends a signal to the potential users of formal methods that we recognise a common scientific method and a common goal - which is to give them the best possible assistance in their ever more difficult task of programming computers reliably to meet their genuine needs. When we have achieved that together, we will not care by what name the method is called, or whether the formulae are enclosed in boxes.

We are not engaged in theological wrangling, let alone religious war. There is no need for a potential user to delay introduction of formal methods until "the experts" have resolved their differences. It is possible even now to gain experience and benefit from any formal method soundly based in mathematics, and to extend the method gradually into all phases of the software lifecycle, taking advantage of new developments and discoveries as they emerge from our research.

That is the message of the title of this conference. I hope that the spirit of that message will permeate the conference, and remain with us when we leave.

Programme Committee

This symposium would not have been possible without the voluntary and dedicated work by all of the PC members:

Micheal Mc an Airchinnigh
Mark A. Ardis
Egidio Astesiano
Stephen Bear
Andrzej Jacek Blikle
Dines Bjørner – Chair
Bernard Cohen
Norm Delisle
David Garlan
Susan Gerhardt
Don Good
Anthony Hall
Anne E. Haxthausen
Ian Hayes
Tony Hoare – Co-Chair
Hans Langmaack – OC Liaison
Peter Lucas
Silvio Meira
Kees Middelburg
Maurice Naftalin
Nikolaj Nikitchenko
Vladimir Red'ko
Dan Simpson
Hans Toetenel
James Woodcock
John Wordsworth

The PC Chairman thanks all and everybody for their patience and forebearance with their chairman.

Organisation Committee

This symposium would not have been possible without the voluntary and dedicated work by the OC members:

Christoph Blaue
Gerd Griesser – Co-Chair
Manfred Haß
Götz Hofmann
Hans Langmaack – Chair, PC Liaison
Uwe Schmidt
Rolf Peter Schultz
Reinhard Völler – Co-Chair

We are sure that all participants at the symposium will rightfully appreciate the hard work this committee has laid down in order to secure a successful symposium.

Acknowledgements

The Programme and Organisation Committees would like to thank, in addition, the following individuals: Rector Magnificus, Prof. Drs. M. Müller-Wille and Prorector Dr. D. Soyka of Christian Albrecht University at Kiel for their support of the holding of the Symposium at Kiel; Messrs. Horst Hünke, Karel de Vriendt and Lodewijk Bos of the CEC for their instigating, perpetrating and enjoyable support of VDM Europe; Dr. Hans Wössner and Ms. Ingeborg Mayer of Springer–Verlag for their continued interest in VDM publications; Ms. Annie Rasmussen of Techn. Univ. of Denmark; Dipl.-Inform. Jens Knoop and Ms. Claudia Herbers of Kiel University; and Ms. Sabine Zacharias of ITK Kiel.

External Referees

All submitted papers — whether accepted or rejected — were refereed by the programme committee (PC) members. Most papers were, in addition, refereed by colleagues.

All papers were refereed by at least three reviewers.

We here list the non-PC referees known to us:

Michael M. Arentoft
Christoph Blaue
Andrzej Borzyszkowski
Richard Bosworth
Hans Bruun
Peter Michael Bruun
Karl-Heinz Buth
Fiona Clarke
John Cooke
Martin Cooke
B.T. (Tim) Denvir
Margaret Gallery
Alessandro Giovini
Michał Grabowski
Klaus Havelund
He JiFeng
R.D. Huijsman
Hardi Hungar
Cliff B. Jones
Burghard von Karger
Jan van Katwijk
Steve King
Jens Knoop
K.C. Lano
Peter Gorm Larsen
Matthew Love
Peter J. Lupton
Hans Henrik Løvengreen
Erich Meyer
Brian Q. Monahan
Mogens Nielsen
Alan Norcliffe
Jan Storbank Petersen
Ben Potter
Gianna Reggio
Marek Ryćko
R.G. Stone
Bernard Sufrin
Andrzej Tarlecki
Paweł Urzyczyn
Morten Wieth

The Programme Committee extends its grateful thanks to the very thorough job done by all referees.

We apologize if, inadvertently, we have omitted a non-PC referee from the above list. To the best of our knowledge the above is accurate.

Sponsors

The symposium would not have been accomplished without the very kind support and financial assistance of the following organisations:

Commission of the European Communities
Deutsche Forschungsgemeinschaft (German Research Council)
State Government of Schleswig–Holstein
Christian Albrecht University at Kiel

as well as the associations and corporations listed below:

British Computer Society, UK
Commerzbank AG, Kiel
CRI: Computer Resources International A/S, Denmark
Daimler–Benz AG, Kiel
Deutsche Bank AG, Kiel
Digital Equipment GmbH, Hamburg
Dresdner Bank AG, Kiel
Dr.-Ing. Rudolf Hell GmbH, Kiel
Gesellschaft für Mathematik und Datenverarbeitung, Bonn
Holsten–Brauerei, Kiel
IBM Deutschland GmbH, Stuttgart
IKO Software Service GmbH, Stuttgart
ITK Informationstechnologie Kiel GmbH, Kiel
Nixdorf Computer AG, Paderborn
Norsk Data GmbH, Mülheim
Philips Communication Industry GmbH, Hamburg
programatic GmbH, Düsseldorf
Provinzial Brandkasse, Kiel
Schmoldt&Axmann, Kiel
Sparkassen- und Giroverband Schleswig–Holstein, Kiel
tfl: Danish Telecommunications Research Lab., Denmark
VAG Willer, Kiel

VDM Europe as well as the organisers of VDM'90 Symposium are deeply indebted to them. Sincere thanks have also to be extended to the numerous helpers in preparing the symposium.

And Now!

Enjoy the proceedings — and contact:

Mr. Karel de Vriendt
Commission of the European Communities
DG XIII, Directorate A
Avenue d'Auderghem 45, Breydel 10/214
B-1049 Bruxelles, Belgium

for further information on VDM Europe.

Table of Contents

Design Methodology

Modularity

Object Orientedness

Processes, Concurrency and Distributed Systems

Formalisations

Type Systems

Foundations

Object Orientation and Formal Techniques
(Extended Abstract)

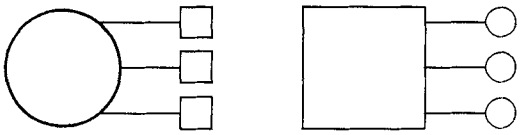
Ole-Johan Dahl
Dept. of Informatics, University of Oslo

1 Introduction

Object orientation, as it appears in Simula 67 [1], was motivated by two main concerns: To achieve good structural correspondence between discrete event simulation programs and the systems being modelled. And to provide language mechanisms for the construction of reusable program components while maintaining good computer efficiency. We shall attempt to view these mechanisms in the light of formal specification and reasoning techniques.

Object orientation has proved to be a successful structuring strategy also outside the area of system simulation. This is due to the fact that objects are useful general purpose tools for *concept modelling*, and can lead to better program decomposition in general, as well as new kinds of reusable program components. It is worth noticing that the class concept of Simula 67 is used to represent "modules" and "packages" as well as object classes. In the following we interpret the term object orientation in this wide sense.

It has been said many times that algorithm and data belong together; the one is meaningless without the other. Together they comprise a dynamic system which exists in time and space. An algorithm needs time to take place and data to operate on; data need space on storage media to exist and associated operations to be accessed and updated. It is reasonable therefore that program components in general should specify data as well as operations. In object oriented programming languages time/space duality is explicit for the two main kinds of program components: a procedure typically interacts with the program environments through data-like parameters, and an object interacts through associated operations. This is illustrated by the following figures, where circles and squares represent (patterns for) operation-like and data-like entities respectively.



Procedure Class

If we abstract away the concepts of time and space we are left with the more purely mathematical notions of *functions* and (timeless) *function applications* and *types* of (spaceless) *values*. In Algol-like languages typed values are of insignificant volume, say, one computer word or less, thus one can afford to implement "spacelessness" by allowing value copying and the storage of anonymous intermediate results at the discretion of the compiler. (In contrast potentially high-volume structures like arrays are not treated as spaceless values, but rather as composite program variables.) It might have been prudent to reserve the term "function" for operations without side effects and taking insignificant time, but that is not usually done.

In the following we assume a strongly typed programming language and use the following terminology: A *procedure* is a pattern for timed action sequences (with associated data), called procedure *activations*. A *function procedure* is a procedure which returns a value and has no other effect. A *class* is a pattern for stored data structures (with associated procedures), called *class objects* (or simply objects if confusion with the non-technical term is unlikely). The data components of an object are called *representation variables*. The associated procedures are called *procedure attributes*. We assume that the representation variables are invisible from outside the object, only the procedure attributes should normally be accessible. The reason for this is twofold:

- The representation variables may be assumed to satisfy an invariant, the *representation invariant*, which is essential for the correct operation of the procedure attributes. By preventing interference from outside one can obtain a kind of correctness guarantee or, rather, if the invariant does get destroyed the error is necessarily located within the object.
- The object as a whole may represent some kind of "abstract" entity, for which the actual representation is or ought to be irrelevant. By hiding it, different representation schemes (classes) for the same abstract concept can be interchanged without hurting the program logic.

It is worth noticing that a simple program variable can be seen as an object with associated operations for assigning and reading its value, the latter usually implicit. Conversely, an object can be seen as a generalized variable, whose value, or "state", is a data record consisting of the values of the representation variables. The procedure attributes are either initializing or updating procedures, or they are "observer functions". The latter are function procedures accessing or computing

aspects of the object state. (Initialization may alternatively be effected implicitly upon object creation by the object's own actions.) We assume in the following that an initializing or updating procedure has no effect outside the object and returns no value. Thus, with this convention a "pop" operation on a stack object S must be expressed by two procedure activations (using the traditional dot notation for referring to procedure attributes):

$v := S.top$	– the top element, *top* is an observer function.
$S.pop$	– pop the stack, *pop* is an updating procedure.

Since objects are potentially high-volume structures, one does not want to implement spacelessness as above by wholesale copying of object-like function values, etc. Typically the state of an object is initialized at the time of its creation and will be updated incrementally through the activation of procedure attributes. If objects occur as parameters to procedures, they are transmitted by pointer rather than by copy.

2 Formal Class Specification

The "interface specification" of a class typically contains syntactic and semantic information about each procedure attribute. The syntactic information may include the procedure name, the number and types of parameters, and the type of function value if any. The semantic information, usually informal, explains the purpose and effect of the procedure. Clearly the correct understanding and usage of a class will depend on the quality of the semantic specification.

Our aim is to hide complexity by constructing program components with simple external specifications, thereby making complex programs more manageable. Therefore the design of a class should to a large extent be determined by the requirement of specification simplicity, and perhaps by the specification tools available. We consider here the use of formal tools for class specification.

It may sometimes be sufficient to use some standard technique for specifying the procedure attributes, like providing pre- and postconditions expressed in some formula language. But these would have to refer to the object state in terms of the representation variables, which means that the kind of abstraction hinted at above can not be achieved in this way. Abstraction may require a complete reinterpretation of objects and their procedure attributes.

2.1 Trace specification

One "abstract" object specification technique, pioneered by D. Parnas, see e.g. [6] or [7], is based on the introduction of *traces* as the means to capture objects as updatable structures. A trace is a sequence of calls for updating procedures, including parameter values. A specification describes the set of legal traces, and defines state observing functions as functions of traces, as well as an equivalence relation on traces.

An advantage of this technique is that it leads to specifications in terms of a set of standard basic concepts and of a uniform style. Furthermore, the trace concept is useful for modelling objects with internal nondeterminacy. The extent to which it lends itself to systematic program design and mechanized reasoning and verification is not clear. Certain object structures like trees must be specified in fairly indirect ways.

2.2 Specification by abstract types

By an "abstract type" we mean a set of values with associated functions, defined algebraically through equational axioms. Its purpose is to be a *tool of reasoning*, possibly aided by automatic or semi-automatic mechanisms for expression simplification and proof construction. Concepts like integers, stacks, tables, lists, trees, and so forth, all with associated operators, can be given simple and succinct definitions which display essential properties in a representation independent way.

Let C be a class, whose objects are fully deterministic internally. Our intention is to relate C to a suitably defined abstract type T, in such a way that T can be seen as a formal specification of C. This is done in two steps, first regarding any C-object as a program variable of a "concrete" type T_C corresponding to the state space of the object, subject to wholesale assignments rather than incremental updating, and then relating T_C to T through an "abstraction function"

$$\mathcal{A} : T_C \rightarrow T$$

for the reinterpretation of concrete object states as abstract values. This was hinted at in [2], but fully explained by Hoare in [3].

Let the representation variables of C be $r_1 : T_1, \ldots, r_n : T_n$, satisfying the representation invariant $R(r_1, \ldots, r_n)$. Thus the object state space T_C is $T_1 \times \ldots \times T_n$, restricted by R. We assume that each T_i is either an implemented type, or is (the state space of) a class not containing C. Let *init*, *upd*, and *obs* be typical initiating, updating, and observing procedure attributes, respectively, with parameter list $u : U$ of ordinary types for simplicity. Assume that these procedures satisfy