FACS Europe

The Newsletter of the BCS Formal Aspects of Computing Science Special Interest Group and Formal Methods Europe.

Series I Vol. 2, No. 3, Winter 1996 ISSN 1361-3103

Contents

Editorial

Welcome to the Xmas Workshop '95 special issue. We had a goodly gathering at Imperial on 18-19 December, seeing out the year in traditional FACS style with a miscellany of mind-broadening topics - this year on the theme of Semantics, both mathematical and meaningful. A special highlight, which unfortunately did not leave a record which we could publish, was the closing panel session, ably and entertainingly chaired by Martin Hyland. (Apologies to Alan Hutchinson, that his amended paper does not appear in this issue. It will hopefully appear in the next issue.)

Delights to come are the Refinement Workshop at Bath in the summer; the Formal Aspects of HCI workshop in September, and of course Xmas 96 - a joint event with the BCS Requirements Engineering SIG. If you have any other ideas for events, please email them to FACS@lut.ac.uk, or talk to any of us...

Contributions Welcome...

Contributions to the Newsletter on any relevant topic are welcome. Please send them electronically, in LaTeX or TeX form if you can; next best is plain ASCII. Otherwise please send A4 copy fit to reproduce by fast photocopying (i.e. no paste-ups), with 300dpi laserprint or equivalent a minimum standard. We will not convert WP formats or type up manuscripts. We will not reproduce extensive notices of events which are also available electronically; please send a short notice (max 1 page) with pointers to more extensive information where available. Please always include a postal or telephone contact for those without email.

Please email to *FACS@lut.ac.uk* or to me or Margaret West at *scomaw@zeus.hud.ac.uk*, *m.m.west@hud.ac.uk*, or alternatively by snailmail to:

FACS/FME Newsletter
c/o Ann M Wrightson
School of Computing and Mathematics
University of Huddersfield
Queensgate
Huddersfield
HD1 3DH
UK

Contributions express the opinions of contributors, not of FACS, FME or any organization with which they are associated (unless they say otherwise!).

Letters are welcome and should be sent to the Editor.

FACS-E Recent books column

Cliff Jones

January 25, 1996

I agreed to produce listings of books which relate to the purpose of this newsletter. Authors should send references in BibTeX format to `cbj@cs.man.ac.uk`; we try to pick up some citations without authors intervention so you can also check via WWW `http://www.cs.man.ac.uk/fmethods/facj/index.html` to see whether your reference has been noted.

References

[And91] James H. Andrews. *Logic Programming: Operational Semantics and Proof Theory.* Distinguished Dissertation Series. Cambridge University Press, March 1991.

[Ban93] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.

[Ban94] U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer Academic Publishers, 1994.

[BFL+94] J C Bicarregui, J S Fitzgerald, P A Lindsay, R Moore, and B Ritchie. *Proof in VDM: A Practitioners Guide.* FACIT. Springer-Verlag, 1994.

[Boe95] Egon Boerger, editor. *Specification and Validation Methods.* Clarendon Press, 1995. ISBN 0-19-853854-5.

[Bow94] J.P. Bowen, editor. *Towards Verified Systems.* Real-Time Safety Critical Systems Series. Elsevier Science Publishers, 1994.

[CD94] S Cook and J Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, Sept 1994.

[Dev90] Yves Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, Wokingham, England, 1990.

[FB94] R. Floyd and R. Beigel. *The Language of Machines: An Introduction to Computability and Formal Languages.* New York: Computer Science Pr., 1994. ISBN 0-7167-8266-9.

[FHMV95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, Mass., 1995.

[FJM94] Loe M. G. Fiejs, Hans B. M. Jonkers, and Cornelius A. Middelburg. *Notations for Software Design.* Springer-Verlag, 1994. ISBN 3-540-19902-0.

[Heh93] E. C. R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.

[JL92] R. Janicki and P. E. Lauer. *Specification and Analysis of Concurrent Systems.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[Lan95] K Lano. *Formal Object-oriented Development.* FACIT. Springer-Verlag, 1995.

[Lau93] P. E. Lauer, editor. *Functional Programming, Concurrency, Simulation and Automated Reasoning.* Lecture Notes in Computer Science 693. Springer-Verlag, 1993.

[LH93] K Lano and H Haughton. *Object-Oriented Specification Case Studies (First Edition)*. Object-oriented Series. Prentice Hall, 1993.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[Mit94] Richard Mitchell. *Abstract Data Types and Modula-2*. Prentice-Hall, 1994.

[Old94] E.-R. Olderog, editor. *Programming Concepts, Methods and Calculi*. North-Holland, 1994.

[Pif91] Mike Piff. *Discrete Mathematics: an introduction for software engineers*. Cambridge University Press, 1991. ISBN 0-521-38622-5.

[Plü90] Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 1990.

BCS-FACS Xmas Workshop 1995

Semantics

Kevin Lano
Imperial College

Ann Wrightson
University of Huddersfield

Andrew Pitts, Andrew.Pitts@cl.cam.ac.uk

Operational Extensionality for Typed Higher Order Languages with State

For higher-order, deterministic, sequential languages with state---such as Scheme, Standard ML, or Algol---there is general agreement that some form of Morris-style contextual equivalence forms a reasonable basis for a theory of program equivalence. The problem is that, with its quantification over all possible contexts, the definition of contextual equivalence is rather intractable. For typed languages, one can hope that at least there is some compositional characterization of contextual equivalence so that, for example, equivalence at a function or procedure type is explained in terms of equivalence at the argument and result types. In this talk, I will summarise what is known about this topic. In a nutshell, the situation for block-structured local state is reasonably good, whilst for dynamically allocated local state it seems quite bad.

Luke Ong, Luke.Ong@comlab.oxford.ac.uk

Game semantics

Game semantics is an unusual denotational semantics in that it captures the intensional (dynamical and algorithmic) aspects of the computation. This talk aims to give an introduction to game semantics for functional computation with particular reference to the so-called Full Abstraction Problem for PCF. We shall survey ideas in denotational semantics motivated by the Problem, and sketch the construction of a fully abstract game model of PCF based on (the category of) arenas and innocent strategies. (Joint work with Martin Hyland.)

Duska Rosenberg (with Keith Devlin), Duska.Rosenberg@brunel.ac.uk

Language at Work

Computer Supported Cooperative Work (CSCW) is a growing field of research which looks at computer-based systems as one of the artefacts used by groups of people working together. One of the important things that happens when people use computers is that information flows from and via computers. The semantics of this information flow is closely related to the role of language in CSCW, a complex pattern which includes communication inside, outside and through a computer. This interplay between an information-rich artefact and human communication is at the centre of 'Language at Work'.

Alan Dix, alan@zeus.hud.ac.uk

From Programs to People: Formal Methods meets the Freedom of the Human Spirit

Program language semantics are, in common with all formal semantics, by definition meaningless. A formal semantics can at best give meaning relative to some context. A formal specification of a sort algorithm has no meaning until it is associated with a real programming language, but of course the language itself only has meaning with respect to a compiler etc. ...

User interfaces are to some extent different. We have a context given to us - the human user. Formal semantics for user interfaces do therefore have natural boundaries: the user input on one side and system display on the other. We can build formal models of this nature and describe some interface usability properties over the models.

Unfortunately even that does not ground the semantics entirely. To really capture the meaning, we need to understand human perception and cognition. But people are so wonderfully unpredictable ... are we on a highway to nowhere, trying to formalise people?

For modelling cooperative work the situation is more extreme. Not only do we have individual psychology, but also social processes at work. Again, one approach is to capture these formally, but perhaps that is not necessary. In cooperative work, the critical things are not so much what people are thinking, but the external representations that they use and their interactions with one another. In some ways formalising cooperative systems may be easier than those for individual users.

* The equivalence problem for semantic data-specifications
Frank Piessens and Eric Steegmans
Frank.Piessens@cs.kuleuven.ac.be

* An Operational Theory of Objects
Andrew D. Gordon
Andrew.Gordon@cl.cam.ac.uk

* Tutorial: Introduction to Situation Semantics
Ann Wrightson
scomaw@zeus.hud.ac.uk

* The Semantics of Garbage Collection Rules, a Denotational Approach
GH.Row@ulst.ac.uk

* Categorial semantics for Object-oriented Specification
Kevin Lano and Jose Fiadeiro and Stephen Goldsack
kcl@doc.ic.ac.uk

* Animation is Approximation
Margaret West
mmwest@scs.leeds.ac.uk

* Semantic Shadowing in the Software Development Process
PJ Lundy & DW Bustard.
p.j.lundy@ulst.ac.uk

Formalizing Pre-conditions as Firing Conditions Using Computations
Andy S. Evans.
a.s.evans@comp.brad.ac.uk

* Existence and Intuitionistic All-Elimination
Alan Hutchinson
alanh@dcs.kcl.ac.uk

Language at Work

Duska Rosenberg*

December 18, 1995

Abstract

This paper presents an analysis of "information bottlenecks" in a real working environment, using situation theory as the analytical framework. Our work is motivated by the need for feedback between groups of human experts engaged in cooperative activities. It is focused on the structure and function of "common artefacts" whose main role in the organisation of human activities is to facilitate information flow. The main aim is to discover under what circumstances and, wherever possible, for what reasons computerised common artefacts create information bottlenecks instead of facilitating interaction, communication and cooperation in the workplace.

Our analysis distinguishes three areas of human expertise and praxis in the application domain (in this case, manufacturing computer systems): the technical expertise of computer engineers, the 'linguistic' constraints that govern communication via a common artefact—the Parts Repair Form PRF—and the social structure and work practices that prevail in the computer industry. Each of these is captured by means of a situation. It is the facts supported by, and the constraints salient in, each of these situations that influence the various activities we seek to investigate.

We note that these three kinds of knowledge are normally considered by analysts from quite different fields: technical expertise is studied by computer scientists and knowledge engineers working on expert systems; linguistic knowledge is studied by linguists; social structure and work praxis is the domain of the social scientist. Each of these disciplines uses distinct techniques and different forms of representation. Situation theory allows us to treat all three kinds of knowledge within the same, uniform framework.

In carrying out an analysis, we use a situation-theoretic methodology as a tool for discovery, not as some kind of specification language. In the discovery process, we are constantly guided by having to seek answers to the questions:

- *What are the relevant constraints?*
- *What are the situations involved, and what are the relationships between them?*
- *What relevant information is transmitted?*

This forces us to adopt both a uniform framework and a consistently high degree of precision, even at stages that might not seem problematic. When a problem is encountered, we 'zoom in' on that part of the analysis, increasing the mathematical precision until a level of detail is reached that is sufficient to provide a resolution to the problem.

*Joint work with Keith Devlin.

E-mail Addresses

Duska Rosenberg: Duska.Rosenberg@brunel.ac.uk
Keith Devlin: devlin@csli.stanford.edu
All CSLI Reports are available as downloadable dvi files from the CSLI World Wide Web entry.

References

[1] Devlin, K. *Logic and Information*, Cambridge University Press (1991, 1995).

[2] Devlin, K. and Rosenberg, D. Situation Theory and Cooperative Action, in Aczel, P. Israel, D, Katagiri, Y. and Peters, S. (eds) *Situation Theory and its Applications, Volume 3*, CSLI Lecture Notes 37 (1993), pp.213–264.

[3] Devlin, K. and Rosenberg, D. Networked Information Flow via Stylized Documents, CSLI Report No. CSLI-94-187 (1994).

[4] Devlin, K. & Rosenberg, D. Layered Formalism and Zooming in the Analysis of Stylized Documents, CSLI Report No. CSLI-94-189 (1994).

[5] Devlin, K. & Rosenberg, D. Interpretive Grammar and Stylized Documents, CSLI Report No. CSLI-95-000 (1995).

[6] Devlin, K. & Rosenberg, D. *Language at Work: Using Situation Theory to Analyze Real Situated Language*, CSLI Pubications, (1996, to appear).

[7] Sacks, H. On the Analyzability of Stories by Children, in Gumpertz, J. & Hymes, D. (eds): *Directions in Sociolinguistics, The Ethnography of Communication*, Holt, Rinehart and Winston Inc. (1972), pp.325–345.

The equivalence problem for semantic data-specifications – extended abstract

Frank Piessens*   Eric Steegmans

September 15, 1995

The first step in the design of a large database is specifying in some sense the part of the real world about which we want to store information in the database. Such a specification is called a *semantic data-specification*. Essentially, the goal of a semantic data-specification is to build a mathematical description of this small part of the real world. This part of the world that interests us, is usually called the Universe of Discourse (UoD) in the database literature. Many formalisms for making semantic data-specifications are in use today, the most prominent ones being Entity-Relationship diagrams ([Ch 76]), and their various extensions. In this paper, data-specifications are defined to be (generalized) sketches, as in [Ca 95, Pi 94]. Entity-Relationship diagrams, and other popular data-specification mechanisms can all be translated to these sketches. Consult [Ca 95] for more details.

The equivalence problem

An important problem in database design is the fact that the same UoD can be described in a number of different, non-isomorphic ways. Two syntactically different Entity-Relationship diagrams can still be descriptions of the same UoD. Consider, as a very simple example, the ER-diagram with one entity class Person, with an attribute Sex, which could be either male or female. Compare this with the ER-diagram where you have two entity classes, Men and Women, and no attributes. These two ER-diagrams, although not isomorphic, do describe the same UoD. This situation is similar to the situation where two syntactically different programs compute the same function, and hence are semantically equivalent. We say that two data-specifications are *equivalent* iff they describe the same UoD. Note that this is not a formal criterium: "describing the same UoD" is an informal notion. In this paper, we consider two different possible formalizations of the notion of equivalence, and compare them.

The fact that the same UoD can be specified in different ways makes the process of integrating a number of existing data-specifications into one large data-specification a very difficult process. Algorithms to decide equivalence can make this integration process much easier (see for instance the discussion of this fact in [Pi 94]). Therefor, we will also discuss decidability of equivalence.

Criteria for equivalence

In the study of process algebras, many different definitions of equivalence of processes are studied: bisimilarity, testing equivalence, behavioural congruence, etc..., and it is not always obvious which equivalence relation must be preferred.

*Research Assistant of the Belgian Fund for Scientific Research

A similar difficulty occurs with semantic data-specifications. There are at least two possible definitions for equivalence:

1. Two data-specifications (considered as sketches) are equivalent iff their theories are equivalent as categories.
2. Two data-specifications (considered as sketches) are equivalent iff their model-categories in **Set** are equivalent as categories.

It should be obvious that the first definition defines a smaller equivalence relation than the second definition. The first definition captures in some sense the intuition that two data-specifications are equivalent iff all the data present in the first one can be computed from the data present in the second one and vice versa. The second definition captures the intuition that two data-specifications are equivalent iff they have the same models (instances). Although the two definitions coincide in many cases, we give examples of specifications which are equivalent by definition 2 and not by definition 1. We also show that both definitions have their advantages and disadvantages, and it remains currently unclear which definition must be preferred.

Decidability of equivalence

Decidability of equivalence, for the second definition of equivalence and for a subclass of data-specifications more or less corresponding to ER-diagrams was investigated in [Pi 94]. In this paper, we strengthen the decidability results from [Pi 94] to encompass a larger class of specifications. We also show that the two definitions of equivalence coincide on this class of specifications.

Conclusion

A denotational semantics for a programming language gives a criterium to decide wether two programs are semantically equivalent or not: they are equivalent iff their denotations are equal.

In a similar way, we have considered the equivalence problem for semantic data-specifications. The two definitions for semantic equivalence can be perceived as definitions of a denotational semantics for data-specifications. For the first definition of equivalence, we define the denotation of a specification (sketch) to be the skeleton of its theory, and for the second definition of equivalence, we define the denotation to be the skeleton of its model category.

We have also summarized the known results, and have proven new results concerning decidabilty of equivalence.

References

[Ba 90] M. Barr, C. Wells. *Category Theory for Computing Science* Prentice Hall International Series in Computer Science, 1990.

[Ca 95] B. Cadish, Z. Diskin. "Algebraic graph-based approach to management of multibase systems, I: Schema integration via sketches and equations." To appear in the proceedings of Next Generation of Information Technologies and Systems, NGITS'95, Naharia, Israel, June 1995.

[Ch 76] P. P. Chen. "The Entity-Relationship Model – Towards a Unified View of Data" *ACM Transactions on Database Systems*, Vol. 1, No. 1,1976,pp. 9–36.

[Pi 94] F. Piessens, E. Steegmans. "Canonical forms for data-specifications", Proceedings of Computer Science Logic 94, Springer Verlag LNCS 933, pp. 397–411.

An Operational Theory of Objects

Andrew D. Gordon
University of Cambridge Computer Laboratory

December 1995

An object calculus is an attempt to capture the essentials of object-oriented programming as a small self-contained language, suitable for theoretical study. Abadi and Cardelli [1] have developed a range of object calculi, mainly in an attempt to provide type theories capable of expressing common idioms of object-oriented programming.

My talk will introduce one of Abadi and Cardelli's calculi and outline an operational theory [2]. The main result is that we characterise contextual equivalence of objects as a form of CCS-style bisimilarity. We use bisimilarity as a tool to justify Abadi and Cardelli's equational theory of objects. In the presence of certain natural typing rules, bisimilarity is the only known model for Abadi and Cardelli's calculus.

This is joint work with Gareth Rees.

References

[1] Martín Abadi and Luca Cardelli. **A Theory of Objects**. 1996. To appear.

[2] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In **Conference Record of the Twenty Third ACM Symposium on Principles of Programming Languages**. ACM Press, 1996. To appear.

"The Semantics of Garbage Collection Rules a Denotational Approach"

By George Row and Averil Meehan
Interactive Systems Centre,
School of Information and Software Engineering,
University of Ulster, Magee College, Derry, N. Ireland, BT48 0ER
email: GH.Row@ulst.ac.uk

Extended Abstract

Abstract of the Abstract
This paper presents and discusses a formal specification of a set of rules for adding garbage collection to the source code of a program written with a naive, infinite-memory view of the machine which will execute it.

Garbage Collection - Combining Correctness and Memory efficiency

The use of Abstract Data Types (ADTs) [Ellis 91, Thomas 88, Guttag 77,78] as the basis of encapsulation and information hiding [Ghezzi 91, Lamb 88, Parnas 72A,72B] is well established. However in languages (such as Modula-2, C or Ada) where the programmer has responsibility for dynamically allocated memory the use of ADT's has been inhibited by the associated memory management problems.

Explicit deallocation of dynamically allocated memory can considerably increase the complexity of a program and introduce the most subtle of errors. Never deallocating, although safe restricts the scale of programs and of the problems to which they may be applied.

The programming style used is also affected. Harrison and Schmidt [Harrison 93] point out the importance of the value delivering style of programming combined with the use of dynamic linked structures. However this style has such severe memory management problems as to cause authors such as Martin, [Martin 86], and Mitchell, [Mitchell 92], to advocate other programming styles. An empirical evaluation [Meehan 93] confirmed the extent of the memory requirement, and as a result, the functional style is the focus of our investigation. However the work is also relevant to the use of procedures with variable parameters.

Extending earlier work by Bilbe [Bilbe 85] to cope with linked structures and value delivering procedures, a simple set of rules for Reference Counting Garbage Collection (RCGC) of ADTs has been developed. (An informal description of these rules is included in Appendix A.) An empirical study [Meehan 93], has shown that when ADT's are used with the RCGC rules, memory is safely recycled. Programs incapable of running within available memory can, with the application of the RCGC rules, run using only a fraction of available memory. The reasoning behind the development of each of these RCGC rules is discussed by Meehan and Row [Meehan 94].

The major advantage of this approach over others, [Mitchell 92] is that no restrictions are placed on either the programming style, or on the data structure used to implement ADTs. The program may be written in a style based on an infinite-memory model and subsequently, when the program has been shown to be correct, the RCGC rules may be applied to produce a correct and memory efficient program.

This leads to the additional advantage that, although the derivation of the RCGC rules was based on an analysis of the dynamics of the implementation of the ADT's, the rules themselves need only be applied to the static text of a program. This gives the RCGC rules the potential for automatic application at compile time. To this end the value of a formal definition of these rules is investigated.

The Formal Specification of RCGC Rules

It was found that a syntactic approach was insufficient to express the RCGC rules, which could only be fully defined by considering the programming language semantics. Unfortunately the techniques and theory for semantic definition are not as developed as syntax definition [Schmidt 86] e.g. there is no standard notation such as the widely used BNF for writing semantics.

Stepney [Stepney 93] points out that the development of a compiler needs as a first step the mathematical definition of both the source and target languages. The derivation of the compiler from these formal semantic definitions results in a reliable compiler whose correctness can be proven. Our problem is rather different but the principle is similar.

The source language is Modula-2 without regard to memory management of ADT's. The target language is Modula-2 with RCGC for ADT's. The transformation rules require more than a syntactic transformation. They are concerned with the static semantics of the language. We formally define both source and target constructs, for a sub-set of RCGC rules.

This clearly shows the processing required to automate the application of these rules. It is hoped that taking this wider approach will provide insight not only for RCGC, but also for other static semantics problems.

Our formal definition is based on denotational semantics [Strachey 66, Milne 76, Tennent 76, Stoy 77, Gordon 79, Tennent 81, Schmidt 86] which maps program code to its denotation using functional calculus as the metalanguage. Our reasons for using denotational semantics are:

(a) its firm mathematical basis [Stoy 77, Milne 76, Scott 76,82] which facilitates not only reasoning about programs but also understanding and development of concepts involved in denoting a wide range of constructs of programming languages [Stoy 77]. Mizuno [Mizuno 92] illustrates the value of this formal basis of denotational semantics by using it to derive, and prove, a security flow control algorithm. Aiken [Aiken 95] commends the increasing use of denotational semantics in the design of programming languages giving precision and correctness to their implementation.

(b) Denotational semantics is not tied to any particular implementation so that our formal definition can easily be adapted for other languages which have the same computational model.

(c) Denotational semantic specifications are especially useful for recursive programs [Pasztor 90]. Moreno [Moreno 92] points out that denotational semantics is suitable for describing functional programming languages as it is higher order. Recursive programs, especially written in the functional style, as well as recursive data structures make the greatest memory demands [Meehan 93, 94] and so they formed the main emphasis of our investigation.

(d) compiler correctness proofs have traditionally been based on denotational semantics [Palsberg 92]

In the light of our experience using denotational semantics, and also that of other researchers, the value of denotational semantics as a definition tool is discussed. Criticisms which have been made of denotational semantics are considered in light of work within the last 10 years to address these.

The discussion concludes that many of the problems of denotational semantics have been addressed in recent years. Our experience of using denotational semantics to specify RCGC shows it has:

(a) clarified our own understanding of proposed constructs
(b) explained the formal design precisely
(c) prepared the way for formal development of a processor based on these rules

The formal nature of denotational semantics along with its lack of implementation details makes our approach easily adapted to other imperative languages with a garbage collection problem. In addition the method used could also be useful for other problems whose solution is concerned with static semantics.

Bibliography

[Aiken 95]
"Safe-A Semantic Technique for Transforming Programs in the Presence of Errors"
Aiken A., Williams J.H., Wimmers E.L., Tech Report, to appear in TOPLAS 95

[Bilbe 85]
"Using the Heap for Modula-2 Opaque Types"
Bilbe, C.R., Journal of Pascal, Ada, & Modula-2, Vol. 4, No 6, PP 24-30, 1985

[Ellis 91]
"Data Abstraction and Program Design" Ellis, R., Pitman, 1991

[Ghezzi 91]
"Fundamentals of Software Engineering"
Ghezzi, C., Jazayeri, M., Mandridi, D., Prentice Hall, 1991.

[Gordon 79]
"The Denotational Description of Programming Languages"
Gordon, M.J.C., Springer-Verlag 1979

[Guttag 77]
"Abstract Data Types and the Development of Data Structures"
Guttag J.V., Comm. ACM, 20, pp 397-404, 1977

[Guttag 78]
"Abstract Data Types and Software Validation",
Guttag J.V., Horowitz E., Musser D.R., Comm. ACM, 21(12), pp1048-64, 1978

[Harrison 93]
"Data Abstraction in Modula-2" Glaser H., Harrison R.
Information and Soft. Tech., 35, 11-12, pp 619-626, 1993

[Lamb 88]
"Software Engineering: Planning for Change", Lamb, D, Prentice Hall,1988

[Martin 86]
"Data Types and Data Structures", Martin, J.J., Prentice Hall, 1986.

[Meehan 93]
"Abstract Data Types with Garbage Collection"
Meehan, A. MSc Dissertation, University of Ulster, 1993

[Meehan 94]
"Guidelines for Reference Counting Garbage Collection"

Meehan, A., Row, G.,
Tech. Report. University of Ulster, 1994

[Milne 76]
"A Theory of Programming Language Semantics", Milne, R., Strachey, C.
Chapman and Hall, 1976

[Mitchell 92]
"Abstract Data Types and Modula-2", Mitchell, R. Prentice Hall, 1992

[Moreno 92]
"Denotational Versus Declarative Semantics for Functional Programming"
Moreno J.C.G., Gonzalez M.T..H., Artalejo M.R.
in Proc's CSL'91, LNCS 626, pp 134-148, 1992

[Mizuno 92]
"A Security Flow Control Algorithm and its Denotational
Semantics Correctness Proof", Mizuno M., Schmidt D.,
Formal Aspects of Computing, 4, pp 727-754, 1992

[Palsberg 92]
"A Provably Correct Compiler Generator", Palsberg, J.,
L.N.C.S., 582, pp 418-434, 1992.

[Pasztor 90]
"Recursive Programs and Denotational Semantics in Absolute
Logics of Programs", Pasztor, A.
Theoretical Computer Science, 70, 1, pp 127-150, 1990

[Parnas 72A]
"On the Criteria to be Used in Decomposing Systems into Modules"
Parnas, D.L., Comms of the ACM, Vol. 15, No 5, PP 330-336, 1972

[Parnas 72B]
"A Technique for Software Specification with Modules", Parnas, D.L.
Communications of the ACM, Vol. 15, No 5, PP 1053-1058, 1972

[Schmidt 86]
"Denotational Semantics", Schmidt, D.A., Allyn and Bacon, 1986

[Scott 76]
"Data Types as Lattices", Scott, D.,
SIAM J. of Computing, vol. 5, pp 522-587, 1976

[Scott 82]
"Domains for Denotational Semantics", Scott, D.,
LNCS Vol. 140 pp 577-613, Springer-Verlag

[Stepney 93]
"High Integrity Compilation", Stepney, S.,
Prentice Hall, 1993

[Stoy 77]
"The Scott-Strachey Approach to Programming Language Theory"
Stoy, J.E., MIT Press, 1977

[Strachey 66]
"Towards a Formal Semantics", Strachey, C., in [Steele 66]

[Tennent 76]
"The Denotational Semantics of Programming Languages"
Tennent, R.D., Comm. of the ACM, vol. 19, pp 437-452, 1976

[Tennent 81]
"Principles of Programming Languages", Tennent, R.D., Prentice Hall 1981

[Thomas 88]
"Abstract Data Types" Thomas, P. Clarendon Press, 1988

Appendix A

Summary of Guidelines for Adapting Programs that Depend on ADT's

1) To assign the value of an ADT expression to an ADT variable use the AssignADT procedure. This applies whether the expression is a variable or a call to a value delivering procedure.

2) Expressions should be only one operation deep, so that garbage collection can be carried out at each step. Intermediate assignment may be needed to achieve this.

Adapting Procedures

3) All value parameters should have the UseADT procedure applied to them at the beginning of the procedure body.

4) Just before the end of the procedure, ReleaseADT should be applied to : (a) all local variables
   (b) all value parameters

5) Global variables and variable parameters require no special treatment within a procedure other than that prescribed by rules 1 and 2

6) Value Returning Procedures
   (a) Any value delivering procedure which has one or more parameters or local variables of the ADT, whether the return value is of the ADT or not, requires a local variable called RESULT of the result type.
   (b) At the end of the procedure, the variable RESULT should contain the value that is to be returned. Rule 4 can then be safely applied and all local variables and value parameter Released, before the value in RESULT is returned.
   (c) If the value returned is of the ADT type, the procedure Prepare should be applied to it before it is returned.

7) Higher Order and Nested Procedures
   As RCGC is encapsulated within the procedure, using procedure types or nested procedures does not require any special treatment.

George Row
Interactive
Systems Centre,
Magee College,
University of Ulster,
Derry, N.Ireland, BT48 7JL

messages: (+44)(0) 1504 375408
Tel: (+44)(0) 1504 375381
FAX : (+44)(0) 1504 370040
email: GH.Row @ ulst.ac.uk

WWW : http://www.iscm.ulst.ac.uk/

Categorial Semantics for Object-oriented Specification

K. Lano, J. Fiadeiro*, S. Goldsack†

December 5, 1995

This paper will outline a semantics for concurrent and real-time object-oriented specification languages such as VDM^{++} and Z^{++} and identify the role played by category-theoretic concepts in providing meaning for inheritance, refinement and subtyping.

1 Introduction

Object-oriented formal specification languages represent a significant contribution to the industrialisation of formal methods, and aim to combine the benefits of precise mathematical notations with the advantages of object-oriented structuring mechanisms. Languages in this field include Object-Z [3], VDM^{++} [4, 7], Z^{++} [7] and MooZ [9].

Because of the newness of the field, there has been more work on development of notation and identifying what capabilities these languages should include, rather than on theoretical foundations. However, recent work has included the development of a denotational semantics for MooZ [8] and an axiomatic semantics for Object-Z [10].

The full paper[1] will provide a mathematical framework which can be used to give an axiomatic semantics for a large part of the VDM^{++} and Z^{++} languages, and discuss the relationships between this framework and that of other formalisms for object-oriented specification and design [1, 5]. A particular concern is the formal definition of subtyping and its properties as an arrow in a category of class specifications.

2 Extended RTL

Our formalism is based on the Real-time Logic (RTL) of [6], with extensions to represent particular method *invocations* and the concept of a general formula holding at a time.

For each class **C** in a specification there is an associated logical language $\mathcal{L}_{\mathbf{C}}$. The meaning of a class **C** is a theory $\Gamma_{\mathbf{C}}$ in its language.

The key features of this language are:

- terms $\clubsuit\mathbf{e}$ where **e** is an *event occurrence* $(\mathbf{E}, \mathbf{i})$, and **E** is an event of the forms $\uparrow\mathbf{m}$, $\downarrow\mathbf{m}$, $\rightarrow\mathbf{m}$ for a method **m** of **C** (initiation, termination and request events), or an event $\theta :=$ **true**, $\theta :=$ **false** for a predicate θ;
- terms $\mathbf{e} \circledast \mathbf{t}$ and $\bigcirc\mathbf{e}$ where **e** is a term, **t** a time-valued term – the value of **e** at **t** and at the next method execution initiation, respectively;
- event counters $\#\mathbf{req}(\mathbf{m})$, $\#\mathbf{fin}(\mathbf{m})$, $\#\mathbf{act}(\mathbf{m})$ for $\mathbf{m} \in \underline{\mathbf{methods}}(\mathbf{C})$;
- formulae $\phi \circledcirc \mathbf{t}$ for formulae ϕ and time-valued terms **t** – "ϕ holds at time **t**";
- modal formulae $\Box^{\tau}\varphi$ "at all future times φ holds", $\diamond^{\tau}\varphi$ "at some future time φ holds", and corresponding versions for "method initiation times": $\Box\varphi$, $\diamond\varphi$ and $\bigcirc\varphi$.

*Dept. of Informatics, Faculty of Sciences, University of Lisbon

†Dept. of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ

[1] Available by ftp from `theory.doc.ic.ac.uk/papers/Lano/bcs95.ps.Z`.

This language, like RTL, supports the specification of safety properties, but also overcomes the deficiencies of RTL in the definition of liveness and fairness properties.

The paper outlines an axiomatic semantics of VDM^{++} using this formalism. It identifies possible alternatives for formalising the *locality* principle of encapsulation: that the state of an object can only be changed by methods of the class to which it belongs. Preservation of locality as an axiom seems to conflict with subtyping, and to lead to excessively restrictive concepts of subtyping which are unlikely to be industrially acceptable. In addition, they have poor category-theoretic properties.

3 Categories

There are three categories which we will consider for object-oriented specification. Each category has classes as its set of objects, but there are (successively weaker) concepts of morphism $\mathbf{f} : \mathbf{C} \rightarrow \mathbf{D}$:

- **Ref**: refinements based on *adequate* retrieve functions from the state of **D** to that of **C**, and surjective total renamings ϕ of the methods of **C** to those of **D**;
- **Sub**: subtypings based on (possibly non-adequate) retrieve functions and (possibly insurjective) total renamings;
- **WSub**: as **Sub**, but with the *frame* or locality requirement for **D** with respect to **C** dropped: that is, new methods not in ran(ϕ) can modify (the interpretation in **D** of) the state of **C**. Such subtyping morphisms are termed *weak* subtypings. They correspond to *invasive superposition* morphisms in [5].

We show that each of these define categories. The first does not possess initial objects or co-products. The second possesses initial objects but not co-products, whilst the final category has a natural co-product construction based on the "disjoint union" of features [5].

Of importance also is the construction of particular forms of pushout which represent repeated inheritance resulting from a common class **A** being inherited via two distinct paths into a class **D**. We give the construction of this pushout in **WSub** and relate it to the pushout of programs defined in [5]. Pushouts are of particular relevance for the integration of separate "viewpoints" in ODP specification [2].

References

[1] Abadi M., Cardelli L.: An Imperative Object Calculus, *Proceedings of TAPSOFT '95*, Springer-Verlag LNCS 915, 1995.

[2] Bowman H., Derrick J., Steen M.: *Viewpoints and Objects*, ZUM '95, LNCS Vol. 967, Springer-Verlag, pp 449–468, 1995.

[3] Duke R., King P., Smith G.: Formalising Behavioural Compatibility for Reactive Object-oriented Systems, *Proc 14th Australian Compt. Sci. Conf. (ACSC-14)*, 1991.

[4] Durr E. and Dusink E.: The role of VDM^{++} in the development of a real-time tracking and tracing system. In Proceedings of *FME '93*, eds. J. Woodcock and P. Larsen, LNCS, 1993, pp. 64–72.

[5] Fiadeiro J. L., Maibaum T.: *Categorical Semantics of Parallel Program Design*, MEDICIS project technical document, Imperial College, 1995.

[6] Jahanian F., Mok A. K.: Safety Analysis of Timing Properties in Real-time Systems, *IEEE Transactions on Software Engineering*, SE-12, pp. 890–904, September 1986.

[7] Lano K.: *Formal Object-oriented Development*, FACIT series, Springer-Verlag, 1995.

[8] Lin T.: *A Formal Semantics for MooZ*, PhD Thesis, DI/UFPE, Recife/PE, Brazil, 1993.

[9] Meira S. and Cavalcanti A.: Modular object-oriented Z specifications. In *Z User Meeting 1990*, Workshops in Computing, pages 173–192. Springer-Verlag, 1991.

[10] Smith G.: A Logic for Object-Z. Technical Report 94-48, SVRC, Dept. of Computer Science, University of Queensland, 1994.

Animation is Approximation[1]
FACS Semantics Workshop, Imperial College, Dec. 1995

Margaret West, University of Leeds: mmwest@scs.leeds.ac.uk

Introduction

An animation is an *abstraction* of a required system and a proof basis for correct animations of Z has been identified [1]. (See [2] for the semantics of Z.) The potential of animation for *explaining* formal specifications is acknowledged and the proof criteria for correctness is *abstract approximation*, based on the notion of *abstract interpretation*. The example interpreter semantics is implemented in a lazy functional programming language. However there are advantages in using a logic programming language for animation purposes: it enables queries of a "what if" variety to be posed. This abstract provides preliminary work on the correct animation of Z using a *Logic Programming Language* (LP) and its *declarative semantics*. The ultimate intention is to implement the declarative semantics of the animation in the LP, Gödel [3]; a pilot study [4] seems promising.

Abstract Interpretation and Abstract Approximation

The seminal work on *abstract interpretation* was done by Cousot and Cousot. (See [5].) In order to capture the underlying structure of a (richer) concrete domain D_{conc}, an abstraction function α is constructed which maps between D_{conc} and an abstract domain D_{abs}. D_{abs} is said to *approximate* D_{conc}. The abstraction function $\alpha : D_{conc} \rightarrow D_{abs}$ and concretisation function $\gamma : D_{abs} \rightarrow D_{conc}$ are monotonic adjoined functions, and $D_{conc}(\sqsubseteq_{conc})$, $D_{abs}(\sqsubseteq_{abs})$, are posets:

$$\forall\, d : D_{abs} \quad d = \alpha(\gamma(d)) \quad \text{and} \quad \forall\, d : D_{conc} \quad d \sqsubseteq_{conc} (\gamma(\alpha(d)))$$

Note that the abstract interpretation is an *upper* approximation where the *top* element corresponds to total lack of information. This is opposite to the usual ordering of domain theory.

In contrast, *abstract approximation* (of the Z notation) is such that the animation abstraction is a *lower* approximation to the concrete interpretation (Z). The animation is an abstracted approximation and the concrete interpretation *refines* the abstract. The evaluation in set-theoretic terms of *schemas, expressions* and *predicates* is termed the $\mathcal{ZF}$ interpretation, in the Z domain *ideal*. (All other parts of Z are expanded out via e.g. the schema calculus.) In order to accommodate non-termination or incomplete information, the sets of *ideal* are 'lifted" by the introduction of a partial element $\perp$ that denotes non-termination. Sets can be *incomplete*, denoted $S_{\cup\perp}$. The ordering relation, $\sqsubseteq$ on *ideal* is equality on integers, and co-ordinatewise on tuples. It uses a standard powerdomain ordering on subsets. The next sections outline preliminary work in using abstract approximation for correct animation in a logic programming language.

Animation Using a Logic Programming Language (LP)

The domain D_Z corresponds to *ideal* and the mappings between D_Z and the abstract (output) domain D_{LP}, are chosen so they are appropriate for the *declarative semantics* of logic programming. For simplicity (as in [1]) the integers, $\mathbb{Z}$, form the basis of the concrete domain D_Z. The logic programming language (LP) is assumed to allow *negation*, and to have built in constructor functions which allow for *list* and *set terms*. Equality (unification) on sets allows for duplication and permutation of elements as in the LP Gödel [3]. It is assumed that the (sequential or parallel) implementation is sound with respect to the semantics.

If [VAR] represent a set of variable names within schemas, Z expressions are evaluated (using set-theoretic considerations) in environment $\rho_Z : VAR \nrightarrow D_Z$. The evaluation of expressions in

[1] A longer version is being prepared as ftp://agora.leeds.ac.uk/scs/doc/reports/1995

D_{LP} involves an environment $\rho_{LP} : VAR \nrightarrow D_{LP}$ and a concretisation function γ is constructed ($\gamma : D_{LP} \nrightarrow D_Z$). If $x : VAR$; $val : D_Z$; $t : D_{LP}$ belong to the appropriate functional domains, then $\gamma(\rho_{LP}(x)) = \gamma(t) = val$ and $\rho_Z = \gamma \circ \rho_{LP}$. γ maps recursively as follows: integers in Z are represented by integers in the LP, sets by set terms, and n-tuples by functions of arity n. The representation of schemas in D_{LP} will be explained further in the next section. Note that for a particular implementation, a computation can fail to terminate while determining the value of a term, so that sets, lists, and so on can be incomplete, or contain elements which are themselves incomplete. The non-termination value $\perp$ in D_{LP} maps to $\perp$ in D_Z.

Evaluation in the LP of Z Expressions, Predicates and Schemas

Syntactic expressions ϵ are interpreted in D_Z with environment ρ_Z using set-theoretic considerations by $\mathcal{E}[\![\epsilon]\!]\rho_Z$. In particular a *schema* evaluates to a set expression, of bindings of variables to values. A schema is interpreted in the LP via. its characteristic predicate: $Schema \Leftarrow A_1 \wedge \ldots \wedge A_s$, where the variables of *Schema* correspond to the declared schema variables and each A_i is an atom. For each declared variable, there is a corresponding atom, declaring the variable type; other atoms interprate the schema predicate. For each $i, 1 \le i \le s$, the predicate P_i in A_i is defined by a set of statements of the form $A \Leftarrow \mathcal{F}$ in the program, where $\mathcal{F}$ is a formula in FOL and A is an atom with P_i. Atoms can either be used to *check* variable values from ρ_{LP}, or, constructively to evaluate them and augment ρ_{LP}.

A goal (query) $?Schema$ has, a (possibly empty) set of answer substitutions. Each of these corresponds to a binding σ for the schema so that $A_1\sigma, \ldots, A_s\sigma$ is a logical consequence of the program. $P(z, x_1, \ldots, x_k)$ could interpret an operation such as set union, where $\{x_1, \ldots, x_k\}$ is a subset of $\operatorname{dom} \rho_{LP}$. Assuming that a correct answer for $P(z, t_1, \ldots, t_k)$ is a binding, $z = t$:

$$\exists\, z (x_1 = t_1) \wedge \ldots \wedge (x_k = t_k) \wedge P(z, x_1, \ldots, x_k)$$

where $t, t_l, \ldots, t_k$ are terms of D_{LP}. An evaluation function $\mathcal{G}[\![\epsilon]\!]$ gives the interpretation in D_{LP} of syntactic expressions ϵ in an environment ρ_{LP}. If z denotes ϵ then $\mathcal{G}[\![\epsilon]\!]$ is defined:

$$[1a] \quad \mathcal{G}[\![\epsilon]\!]\rho_{LP} = t \Leftrightarrow (z = t) \wedge (x_1 = t_1) \wedge \ldots (x_k = t_k) \wedge P(z, x_1, \ldots, x_k)$$
$$[1b] \quad \mathcal{G}[\![x_i]\!]\rho_{LP} = t_i \Leftrightarrow (x_i = t_i) \wedge true, \quad (x_i \in \operatorname{dom} \rho_{LP}).$$

In order to prove correctness it is necessary to show that the interpretation in D_{LP} is built recursively for each operator of Z represented in (1a, 1b). Recalling that $\mathcal{E}[\![\epsilon]\!]\rho_Z$ interprets ϵ in D_Z, criteria also include the following approximation rule:

$$\gamma(\mathcal{G}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{E}[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

For example evaluating "$2x$" where x has value n, in an implementation where $2n$ exceeds the largest integer available in the system may result in non-termination: $\perp = \gamma(\perp) = (\gamma(\mathcal{G}[\![2x]\!]\rho_{LP})) \sqsubseteq 2n$ (the value obtained when interpreting in D_Z, the concrete interpretation).

References

[1] P T Breuer and J Bowen. Towards Correct Executable Semantics for Z. In *Z User Workshop, Cambridge, June 1994*, pages 185–209. Springer-Verlag, 1994.

[2] S. M. Brien and J. E. Nicholls. Z Base Standard. Technical Report PRG-107, Oxford University Computing Laboratory, Oxford, Nov 1992.

[3] P M Hill and J W Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[4] M M West. Types and Sets in Gödel and Z. In *ZUM'95 - 9th International Conference of Z User's, September 1995, Limerick, Ireland*, 1995.

[5] P Cousot and R Cousot. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13:103–179, 1992.

Semantic Shadowing in the Software Development Process

P.J. Lundy & D.W. Bustard
School of Information & Software Engineering, University of Ulster, Cromore Road, Coleraine, BT52 1SA, Northern Ireland
E-Mail: pj.lundy@ulst.ac.uk & dw.bustard@ulst.ac.uk

Introduction

Although formal methods have been used successfully in a number of specific application areas, such as in the development of safety critical systems and in the definition of international standards, progress towards their acceptance as a routine aspect of all software development still seems some way off. This paper describes work based on the premise that such general use is desirable. The strategy involved has been to examine ways of reducing associated costs on the assumption that the cost-benefit balance can eventually be brought down a level where formal modelling becomes cost-effective.

Method Integration

The basic approach taken is one of *method integration* [1], in which formal techniques are introduced in a supporting role to an existing software development process. Figure 1 shows the general scheme with respect to the V-life cycle model [2] – a typical software development process.

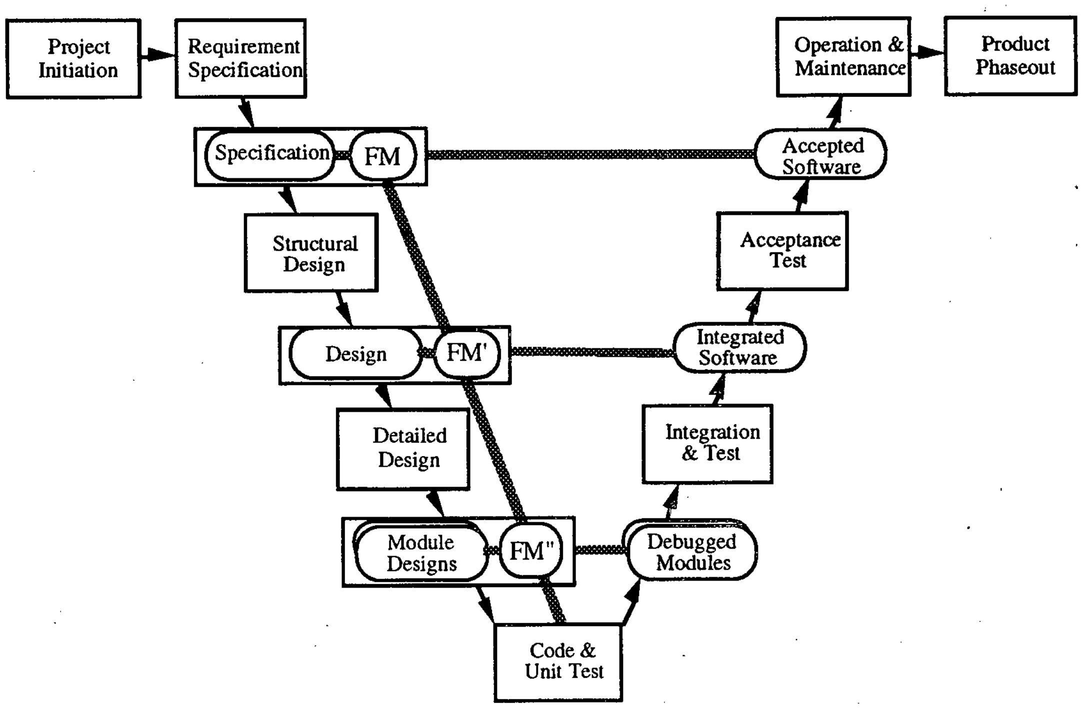

Figure 1: The V-Life Cycle with Formal Modelling

In this life cycle, the development process moves through a series of *phases* (shown as rectangles), each generating a *phase product*. The left hand-side of the V is concerned with analysis and design, while the right hand-side covers implementation to satisfy the specification and design. Each phase product, such as the 'specification', has a base description and, optionally, one or more associated formal models. The thick lines in the diagram imply a need for consistency among the phase products that they link. Thus, for example, the accepted software must match its specification,

including any formal models of the system specified. Also, if formal models are created at various stages of the process these too must be consistent.
The purpose of a formal model in each case is to (i) improve the quality of the base description by highlighting inconsistencies and promoting a greater understanding of what is being defined; (ii) provide a more precise reference description against which the next development phase can be undertaken; and (iii) provide a more precise definition against which the implementation can be verified.

Cost - Benefit Considerations
The level and use of formal models within any instantiation of this development process will be dictated by cost-benefit considerations. Thus, for example, it might be decided that the greatest gain is at the requirements specification level and that the cheapest approach is to use a consultant to build suitable formal models. Alternatively, it might be believed that the greatest gain is in the documentation of software designs and introduce formality at that level as a general programming aid.
Regardless of where formality is *potentially* of greatest benefit, costs must be kept down if it is to be used at all. This means making formal models as easy as possible to construct and maintain.

Deriving A Formal Model Of Requirements.
The earliest opportunity in the life cycle to benefit from formality is by producing a formal model of requirements and, as discussed above, it is desirable to integrate this with the existing approach to requirements engineering. In this work the integrative approach is used to give a precise semantic interpretation to informal models in the RACE requirements engineering method [3]. RACE is currently under development at the University of Ulster and involves the integrated use of business and computing analysis. In essence, the business analysis sets the context for a computing system. Informal behavioural models are developed through the business analysis and formality offers the opportunity to clarify the business system and also strengthen the link to the computing analysis phase that follows. The basic method [4] facilitates the initial building of formal descriptions in LOTOS [5] from informal models and on-going work is addressing attempts to deal with the more difficult problem of maintaining formal models as the system involved changes [6]. Plans for future work are also outlined.

Refs.

1. Semmens LT, France RB and Docker TWG, Integrated Structured Analysis and Formal Specification Techniques Computer Journal, 35 (6), pp. 600-610, 1992
2. Rook P, Controlling Software Projects, Software Engineering Journal, pp. 7-16, 1 (1), Jan 1986
3. Bustard DW: Progress Towards RACE, a 'Soft-Centred' Requirements Definition Method, 1st IFIP/SQI International Conference on Software Quality and Productivity, Hong Kong, pp. 29-36, Chapman & Hall, Dec 1994
4. Bustard DW and Lundy PJ: Enhancing Soft Systems with Formal Modelling, RE '95, IEEE International Symposium on Requirements Engineering, York, pp. 164-171, March 1995, IEEE Computer Society Press
5. Bolognesi T and Brinksma E: Introduction to the ISO Specification Language LOTOS, Computer Networks and ISDN Systems Vol. 14, Jan 1987, pp. 25-59
6. Bustard DW and Lundy PJ: Integrating Process Modelling and Soft Systems Analysis, to be presented at the Methods Integration Workshop, Leeds, March 1996

Formalizing Pre-conditions as Firing Conditions Using Computations

Andy S. Evans.
Department of Computing,
University of Bradford,
West Yorkshire, BD7 1DP.
email a.s.evans@comp.brad.ac.uk

November 1, 1995

1 Extended Abstract

Recently, there has been growing interest in the use of state based notations such as Z for the specification of reactive and concurrent systems [1, 2, 3, 4, 5]. Central to much of this work is the assumption that pre-conditions may be viewed as *firing conditions*. That is, the pre-conditions of an operation may be thought of as defining the conditions that will cause the operation to execute or 'fire' - outside these conditions the operation is impossible. By representing the events of a reactive or concurrent system in this way, one is able to model the eventual and parallel excecution of the events of the system.

Unfortunately, the firing condition interpretation of pre-conditions goes directly against their established meaning in Z. Z specifications concentrate on the 'static' system behaviour. This is why they define operations using state before and after. Furthermore, an 'interpretation' provides a poor basis on which to gain a formal understanding of other aspects of 'firing conditions' such as refinement and proof.

The aim of this paper is to examine ways in which firing conditions can be formalised in Z, *without* having to extend the notation in any way, and which also allows for a separation of concerns between the static and dynamic properties of the specification.

Our approach is very simple: we specify the static behaviour of a concurrent or reactive system in the traditional way using state and operation

schemas. In order to specify 'firing' behaviour, we show how this specification can be extended with a *computation specification*, an additional Z specification which formalizes the execution of the specification in terms of the allowable state-transitions that the system may partake in. We provide some generic definitions which allow a computation specification to be straightforwardly generated for any Z specification of the state and operations of a concurrent or reactive system. Because this specification is also written in standard Z, we show that there is no need to extend the semantics of Z to model concurrent systems.

Finally, we show how to extend the approach with 'fairness' constraints requiring the eventual execution of enabled operations and look at ways in which divergance can be expressed within the model. We also report on work in progress to develop proof and refinement techniques based upon the firing condition approach to specification.

References

[1] M.B. Josephs. *Specifying Reactive Systems in Z.* PRG Technical Report, PRG-TR-19-91, Oxford University Computing Laboratory, 1991.

[2] A.S. Evans. *Specifying and Verifying Concurrent Systems in Z*, Procs. Formal Methods Europe '94, Springer Verlag, 1994.

[3] A.S. Evans, *Z for Concurrent Systems*, PhD Thesis, In Preparation.

[4] B. Strulo. *How Firing Conditions Help Inheritance*, Procs ZUM'95, Springer Verlag , 1995.

[5] S. Stepney, *Testing as Abstraction*, Procs ZUM'95, Springer Verlag , 1995.

BCS FACS
Department of Computer Studies
Loughborough University of Technology
Loughborough, Leicestershire
LE11 3TU
UK
Tel: +44 1509 222676
Fax: +44 1509 211586
E-mail: FACS@lut.ac.uk

FACS Officers