**EUROPEAN WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS**

**TECHNICAL COMMITTEE 7**

**(Safety, Reliability and Security)**

# Guidance on the use of Formal Methods in the Development and Assurance of High Integrity Industrial Computer Systems

## Part III A Directory of Formal Methods

Edited by

S O Anderson, R E Bloomfield and G L Cleland

**Working Paper 4002**

14, June 1998

## Summary

This document contains an evolving directory of industrially used formal methods. It contains a summary of the methods included and detailed entries for each of the methods that describe the application of the method, its maturity, the availability of tools etc. Method users are encouraged to comment on the directory and to propose additions. The format for additions is defined in an appendix and contributions should be emailed to reb@adelard.co.uk. A Word document containing the skeleton format may be downloaded from the URL:  http://www.dcs.ed.ac.uk/ewics/fmdir.htm

# EWICS TC 7

## Formal Methods Sub-group

This document has been compiled by the Formal Methods sub-group of EWICS TC7. Contribution, both great and small have been made by many who have both attended the quarterly TC meetings, and who have reviewed drafts and written new text between meetings. Active members of the Formal Methods sub group during the period of preparation of this guideline are:

| | | |
|---|---|---|
| Dean Ayres | AEA Technology | U.K. |
| Stuart Anderson | University of Edinburgh | U.K. |
| Reinhold Bariess | Univ. Darmstadt | Germany |
| Robin Bloomfield | Adelard | U.K. |
| John Brazendale | Health and Safety Executive | U.K. |
| Andy Coombes | University of York | U.K. |
| G L Cleland | University of Edinburgh | U.K. |
| Jan Gayen | T.U.Brauschiwig | Germany |
| Christopher Gerrard | Gerrard Software | U.K. |
| Peter Daniel | GEC-Marconi Security | U.K. |
| Peter Froome | Adelard | U.K. |
| Geoff Duke | ICS PLC | U.K. |
| Edward Fergus | Health and Safety Executive | U.K. |
| Wolfgang Ehrenberger | Fachochschule Fulda | Germany |
| Peter Froome | Adelard | U.K. |
| Janusz Gorski | Technical University of Gdansk | Poland |
| Bill Johnson | Du Pont | U.S. |
| Fiona Maclennan | Lloyds Register | U.K. |
| Vic Maggioli | Feltronics | U.S.A. |
| Michael Marhoefer | Siemens | Germany |
| Meine van der Meulen | Simtech | Netherlands |
| Elisabeth Noe-Gonzales | Merlin Gerin | France |
| Keith Purves | ERA Technology | U.K. |
| Johannes Reiner | Federal Test and Research Centre, Arsenal | Austria |

## Contents

## 1 Introduction

This section of the guideline is a directory of formal methods. No such directory could ever hope to be fully comprehensive[1]. The approach taken here is to try to cover those methods in common use. We intend that the directory will grow over time and that this draft of the Directory will be bootstrapped by the addition of other methods. If you would like to add an entry for a formal method you have experience of in an industrial context please see Appendix A. The criteria for inclusion in this directory are that a method has been used in some industrial applications and that it provides the means to describe some computational system, express properties of the systems and verify that properties hold of systems.

Providing a meaningful classification of formal methods is difficult because there is a wide range of possible criteria upon which the classification could be based. Possibly the most useful is the application area of a method - but the available data on the application of formal methods is scant and poorly co-ordinated. An attempt is made in Appendix B to provide a classification based on the theoretical basis of methods. This provides some basis for comparison but is not strongly oriented to industrial application.

## 2 How to use this Directory

This Directory contains information about a number of formal methods in the following structure:

*Method:* The title of the method.

*Summary*: A brief summary of the method, its strengths and weaknesses.

*Applications*: A summary of the main areas of application.

*Properties:* A discussion of the kinds of properties which can be described by the method and the ways of reasoning about them.

*Relation to other formal methods*: Closely related methods are described and contrasted with radically different methods.

*Theoretical basis:* A fairly non-technical description of the kind of computational model used by the method and how one goes about specifying properties of computations.

*Tools:* A summary of available tools, their function, robustness, scope, strengths and weaknesses

[1] The world wide web has a number of industrial formal methods sites. One attempt to index these can be found at the URL: http://www.csr.ncl.ac.uk/projects/FME/InfRes/applications. An index of formal methods tools can be found at the URL: http://www-formal.stanford.edu/clt/ARS/ars-db.html. A general resource for formal methods can be found at: http://www.comlab.ox.ac.uk/archive/formal-methods.html

*Appraisal*: A brief introduction to the five appraisal points given below.

*Maturity*: An assessment of how technically mature the method is and its maturity in terms of process, i.e. harmonising with other software development techniques.

*Availability*: An assessment of the availability of consultancy, textbooks and introductions, training, courses etc.

*Strength:* An assessment of the kinds of areas the methods is strong in expressing requirements/properties and where it is weak.

*Industrial Experience:* A summary of experience pointers to specific projects and companies which have experience of the method. If there are case studies in the open literature then these are identified.

*Tool Availability:* The availability of tools is assessed and names of suppliers in the case of commercial tools or institutions developing systems otherwise are given. There is an attempt to be comprehensive here and to avoid commercial bias.

*Bibliography:* Each entry contains some reference material which provides further background to the entry. In addition some entries also have pointers to world wide web or ftp sites which contain further information on the methods

In addition we have tried to provide two matrices that summarise the type of applications that have been undertaken with the method and the roles of tools.

If you are new to formal methods then you may need find the overview of the entries in Table 1 useful The table contains a brief summary and description of the main areas of application of each of the methods in the directory. The full entries contain much more information. The table should help you eliminate a number of methods which are clearly unsuitable for your organisation, you should then consider the remaining candidates by looking at the detailed entries.

**Table 1: Overview of methods**

| **Method** | **Summary** | **Applications** |
|---|---|---|
| CCS | provides the means to model concurrent and distributed systems and has an associated logic which can express properties of systems. | Mature use in protocols verification, use in modelling and reasoning about safety systems and hardware. |
| COLD | State-based, algebraic specification language with | Embedded systems is the main industrial use to date. |

| | | |
|---|---|---|
| | industrial strength tools | |
| OBJ | general purpose formal specification language, algebraic, allows abstraction away from detail | mature in many “data driven” application areas, including GUI, Secure and Safety critical systems, communications |
| SAGA | based on the LUSTRE synchronous data flow language, can model timing and concurrency | reactive systems, real-time embedded systems |
| Z | powerful and flexible notation based in set theory oriented to providing functional models of systems mostly used for specification | general purpose, operating system components, databases, security, hardware, graphics |
| Raise | Model-oriented and algebraic specification language | Main use is in high-reliability applications in aerospace and transportation. |
| B Method | powerful general purpose set theory based system with support for reasoning about implementations and with tool support for proof | railway signalling and train control |

## 3 CCS - a calculus for communicating systems

A calculus for communicating systems[2] and its associated logic - the modal mu calculus.

Summary

CCS provides a way of describing the dynamic behaviour of a system. It does this by giving the designer the ability to specify the behaviour of the system as potential sequences of atomic (or primitive) actions. The logic associated with the system is capable of capturing subtle properties like integrity

Applications

Very mature use in the specification of protocols. The LOTOS protocol specification language is based on CCS. LOTOS also includes the means to specify data types via algebraic specification methods. Recent work includes specification and design support for the implementation of distributed information systems [6]. A recent book describes the role of CCS in the development of distributed systems [5].

Properties

The CCS calculus allows the specification of agents which are intended to capture the dynamic behaviour of systems. The basic notion at the core of CCS is equivalence of agents. This notion can be used to explore the relation between abstract specifications of behaviour and implementations of the specification. The logic used with CCS is very expressive and centres on expressing requirements on the behaviour of systems. In particular one can express the avoidance of undesirable behaviour and the eventual occurrence of required events. Little has been done on the treatment of data dependent properties.

Relation to other formal methods

Closely related to the family of process algebras and various automata-based approaches[3]. In taking the idea of behaviour as central CCS displays a lose connection with other process-based methods e.g. CSP, ACP, Meije and other process calculi. The connection with data-centred approaches like algebraic and type-theoretic methods is less clear. Some methods e.g. LOTOS attempt to combine process and data-centred techniques.

Theoretical basis

The basis is the notion of labelled *transition system* which aims to capture the potential action of systems. A simple example of a labelled transition system is a finite state automaton. The related

[2] Information on tools to support CCS can be found at the URL: http://www.dcs.ed.ac.uk/home/cwb

[3] Statecharts is probably the most widely used method in an industrial context. Details of statechart tools can be found at the URL: http://www.ilogix.com

logical system expresses properties of such systems. Such properties can be seen as formalised requirements.

Tools

There are no industrial strength tools though a large variety of academic tools exist. These fall into two categories: algebraic manipulators which aim to aid symbolic reasoning and model checkers which use state exploration techniques. These are now capable of analysing systems with large numbers of states.

Appraisal

CCS provides a powerful method for modelling systems. The notion of equivalence is capable of relating seemingly quite different agents. The logical system is also powerful and elegant though it takes some time to acquire skill in formulating appropriate properties in the logic.

*Maturity*: Mathematically highly mature, good experience in protocols, distributed systems and hardware.

*Availability*: Few industrial courses available, there are excellent textbooks and a range of supporting expository works are available.

*Strength*: Capable of expressing very subtle requirements of systems and detecting difficult problems in developing implementations from high-level specifications. Weak on reasoning about data and linking to data oriented formal methods.

*Industrial Experience*: Has been used on a number of projects, Praxis UK has used CCS in the development of a large scale distributed system. The LOTOS community experience is relevant to use of CCS. There is little formal evaluation of the use of CCS in the open literature.

*Tools*: Currently there are no commercial tools. There are reliable academic tools which can deal with medium sized systems - dealing with large systems is still a research topic. Medium sized models are useful for reasoning about specifications and specific features of systems.

Case Study Matrix:

| | Formalization | Rigorous Proof | Formal Proof |
|---|---|---|---|
| Code | Milner’s book [1] provides an example formalisation of a simple parallel while language and some rigorous reasoning about programs. | | In two papers Walker [7, 8] considers using the Edinburgh concurrency workbench to show properties of various mutual exclusion algorithms. |

| | | | |
|---|---|---|---|
| System Models | [2] contains examples. | [2] contains examples. | Bruns and Anderson [2] consider a system model of a safety-critical communication protocol and show properties hold of he system using the concurrency workbench. |
| Requirements | | | [5] considers the use of CCS in a variety of realistic applications including requirements analysis. |

Tools Matrix:

| | Requirements | System Models | Code |
|---|---|---|---|
| Commercial | None | None | None |
| Non-commercial | The Edinburgh Concurrency Workbench [3] and the TAV [4] from Aalborg in Denmark provide so called “model checkers” which provide the means to check a property holds of a system are useful for reasoning about requirements, system models and code. | | |

Bibliography

[1] R. Milner, Communication and concurrency, Prentice-Hall, London, ISBN 0-13-115007-3, 1989.

[2] G. Bruns and S. Anderson, The formalisation and analysis of a communication protocol, Formal Aspects of Computer Science, 6, 1, pp. 92-112, 1994

[3] F. Moller, The Concurrency Workbench, LFCS Internal Report, University of Edinburgh, Department of Computer Science, 1990

[4] K.G. Larsen, The TAV tool, University of Aalborg, 1991.

[5] G. Bruns, Distributed Systems Analysis with CCS, Prentice-Hall, 1997.

[6] G. Bruns, Refinement and dependable systems, 10th COMPASS, pp. 49-56, 1995.

[7] [W1] D. Walker, Analysing mutual exclusion algorithms using CCS, Edinburgh University, Department of Computer Science, ECS-LFCS-88-44

[8] [W2] D. Walker, Automated analysis of mutual exclusion algorithms using CCS, Edinburgh University, Department of Computer Science, ECS-LFCS-89-91

## 4 COLD - a Common Object-oriented Language for Design

*Summary*

COLD[4] is a single language unifying the features of a specification, design and programming language. There are two main versions: COLD-K, with orthogonal language constructs, explained in the text book [1] (this version is fine for a basic course on specification at Universities). In the user-oriented language COLD-1 specifications are organised according to helpful patterns with appealing keywords, and several mechanisms are available to manage the name-space of the user-defined identifiers, as explained in the text book [2] (this version is fine for application-oriented courses and for applications in real projects).

*Applications*

Industrial within Philips for embedded software of mass consumer electronics systems in combination with the SPRINT method [3]. Experimental and educational: many case-studies have been published, including text-editing [1] pp. 213-239 and [1] pp. 217-405, database query languages INGRES [1] pp. 171-198 and NDB [5], a copy-management system, a component-placement machine [6], shadow-mask design [7], and a railway-safety system [2] pp. 125-200.

*Properties*

For design in the small, the language supports various styles of description such as algebraic specifications, axiomatic specifications, inductive definitions, pre- and post-conditions, invariants, abstract algorithmic descriptions, as well as executable specifications. For design in the large the language provides constructs for modularization, parameterization and design. It supports the notion of a software component as a basic building block for designs. Some of the characteristic features that distinguish COLD from other wide-spectrum languages such as VDM are its algebraic view of states (“states as algebras”) and the strong focus on techniques for design in the large.

*Relation to other formal methods*

The language is influenced by VDM and by algebraic specification languages. VVSL is partially based on the same theoretical basis. In [2] a formal connection is established with graphical notations (HOOD, Venn-diagrams, SKL, dataflow diagrams etc.).

*Theoretical basis*

For a survey see [8]. Three novel systems of logic were devised specially for COLD:

- the logic $MPL_\omega$ [9] (based on Scott’s E-Logic [10]),
- class algebra [11] (related to module algebra [12]),
- the calculus $\lambda\pi$ (see [4] pp. 88-141 or [13]).

[4] Information on COLD may be found at the URL: http://www.cs.rug.nl/~rix/cold.html

There exists a formal definition of the semantics: [14]. The language includes dynamic logic [15] and inductive definitions [16].

*Tools*

For COLD-K there exists a typechecker. For COLD-1 there is a typechecker called TYCOON and a standard library called IGLOO. For SPRINT [3] there is amongst others a code generator which accepts the language PROTOCOLD, which is a subset of COLD-1. All tools run on SUN-3 and SUN-4.

*Appraisal*

*Maturity:* 5 years work on foundations (1983-1988), more than 9 years work on industrialisation.

*Availability*: text books [11, 12] and [4] are readily available. Tools are owned by Philips.

*Strength:* both a theoretical basis and a user-oriented version (COLD-1) exist.

*Industrial Experience:* embedded software of mass consumer electronics systems.

*Tool Availability*: contact Philips

*Applications and Experience Matrix:*

not available

*Tools Matrix:*

not available

*Bibliography*

[1] L.M.G. Feijs, H.B.M. Jonkers, “Formal specification and design” *Cambridge Tracts in Theoretical Computer Science 35*, Cambridge University Press (1992)

[2] L.M.G. Feijs, H.B.M. Jonkers, CA Middelburg “Notations for software design” Springer-*Verlag FACIT series* (1994).

[3] H.B.M. Jonkers, “An Overview of the SPRINT method”, in: J.C.P. Woodcock, K.G. Larsen (Eds), FME’93: Industrial-Strength Formal Methods, *Springer-Verlag LNCS* 670, pp. 403-427 (1993).

[4] L.M.G. Feijs, “A formalisation of design methods: a λ-Calculus approach to system design, with an application to text editing. *Ellis Horwood* Limited (1993).

[5] L.M.G. Feijs, “Norman’s database modularised in COLD-K”. in: J.A. Mergstra, L.M.G. Feijs (Eds), Algebraic Methods: Theory, Tools and Applications Part 11. Springer Verlag *LNCS 490* pp. 205-231 (1991).

[6] A. de Bunje, L.M.G. Feijs. "Formal specifications applied in industry: a case study using COLD", *Proceedings of the third international workshop on software engineering and its applications, Toulouse,* EC2 (269-287 rue de la Garenne, 92024 Nanterre Cedex France) ISBN 2-906899-48-8, pp. 649-669 (Dec. 1990).

[7] F.J. Van der Linden, "Specification in COLD-1 of a CAD package for drawing shadow masks", in: A. van Lamsweerde, A. Fugetta (Eds.), ESEC'91, *Springer-Verlag* LNCS 550 pp. 101-121 (1991).

[8] L.M.G. Feijs, "An overview of the development of COLD", in D.J. Andrews, J.F. Groote, C.A. Middelburg (Eds), Semantics of specification languages, pp. 15-22, *Springer-Verlag* (1993).

[9] C.P.J. Koymans, G.R. Renardel de Lavalette, "The Logic MPL. in: M. Wirsing, J.A. Bergstra (Eds.), Algebraic Methods: Theory, Tools and Applications, *Springer-Verlag LNCS 394,* pp. 247-282 (1989).

[10] D.S. Scott, "Existence and description in formal logic", in R. Schoenman (Ed.), Bertrand Russell, Philosopher of the Century, Allen & Unwin, London, pp. 181-200 (1967).

[11] H.B.M. Jonkers, "Description albegra", in: M. Wirsing, J.A. Bergstra (Eds.), Algebraic Methods: Theory, Tools and Applications, *Springer- Verlag LNCS 394,* pp. 283-305 (1989).

[12] J.A. Bergstra., J. Heering, P.Klint, "Module algebra,", *JACM* Vol. 37 No. 2, pp. 335-372 (1990).

[13] L.M.G. Feijs, "The calculus $\lambda\pi$ " in: M. Wirsing, J.A. Bergstra (Eds.), Algebraic Methods: Theory, Tools and Applications, Springer- *Verlag LNCS,994,* pp. 307-328 (1989).

[14] L.M.G. Feijs, H.R.M. Jonkers, C.P.J. Koymans, G.R. Renardel de Lavalette, "Formal definition of the design language COLD-K". Revised Edition, ESPRIT document METEOR/17/PRLE/7 (Oct. 1987).

[15] D. Harel, "Dynamic logic", in: D. Gabbay, F. Guenther (Eds.), *Handbook of philosophical logic,* Vol. II pp. 497-604, D. Reidel Publishing Company, ISBN 90-277-1604-8 (1984).

[16] C.P.J. Koymans, G.R Renardel de Lavalette, "Inductive Definitions in COLD-K". *Logic Group reprint series No. 50,* Department of Philosophy, University of Utrecht (1989).

## 5 OBJ - an Executable Algebraic Specification Language.

(Alternatively referred to as an 'Abstract Data Type' or 'ADT' language).

*Summary*

OBJ[5] is a general purpose Formal Specification language which enables users to think about and specify the essence of a system or model in simple algebraic terms, without concern for lower-level implementation detail. Specifications are directly executable.

*Applications:*

OBJ has been used in many diverse applications including the specification of: GUI standards; High-integrity Software Tools; Secure Systems. Safety-critical Systems; Protocols; Communication Systems. Programming Language Semantic Definition.

OBJ seems best-suited to data-driven applications, where the notion of ADT has a 'natural' fit with its real-world counterpart. However, all systems can be thought of in some sense as 'data driven'. So, if you can identify data and actions to perform thereon, then you can go about specifying (or modelling) the system in OBJ, [7, 3, 10].

*Properties:*

There are two parts to the language:

*Theorems* which are non-animatable and specify in precise abstract terms what is required from an operator or system transition, and

*Constructive Specifications* providing an animatable specification of the intended system.

Users typically develop the Constructive Specification, which since it can be 'brought to life' via tool-supported animation, leads to fast and early validation feedback and specification iteration. Then, depending upon the criticality or degree of rigour demanded for the application, go on to define the key Theorems to tie down required behaviour, safety- or security-critical invariants etc. (It is not uncommon for applications to be developed against the Constructive Specification alone - i.e. without recourse to the, generally more intellectually challenging, Theorem definition).

Both Theorems and Constructive Specifications can be (and usually are) refined. Typically an initial Constructive Specification is devised at a high level of abstraction. This introduces Abstract Data Types (i.e. named types, but without any implementation make-up details) and operators acting upon these types. Subsequent levels of refinement may seek to bring out internal make-up of the data types, together with further detail as to algorithms to be employed. Relation to other formal methods: Industrial users have developed a 'fit' with the MALPAS Program Analysers [1].

[5] Information on the family of OBJ methods can be found at the URL: http://www.cs.ucsd.edu/users/goguen/sys/obj.html. Further information can also be found at the URL: http://lex.ucsd.edu/semcomp

This approach has been found to provide a practical rigorous development route from Specification to End-System, with a high degree of V&V support from Commercially Available Toolsets.

LOTOS uses a derivative of OBJ called ACT One for its data dependent modelling.

OBJ Theorems use Z-Like constructs with existential and universal quantifiers and first order logic.

The output of the Formal Specification exercise can be a Functional Specification comprising exact procedure or function signatures coupled with defining pre-, and post-conditions as per the VDM approach.

Another output is a comprehensive suite of end-system test cases to probe for implementation conformance to specification. These test cases are the end-system counterpart of specification 'verification' tests developed early in the lifecycle, before the move to detailed design. [4, 5, 6]

OBJ is well-suited to the specification of 3GL or OO-based systems. [10]

*Relation to other formal methods:*

*Theoretical basis:*

OBJ uses basic algebra. A Constructive Specification is composed from modules, each module introduces ADT's (Abstract Data Types - the elements of the algebra) and Operators which act upon the ADT'S. Operator action is defined in terms of equations, whereby the left-hand-side of an equation is a 'start' point for the action, the right-hand-side gives the result of the action.

For example, we might have an operator defined by:

$$\textit{Safe to open?: Valve-id, Pipe} \rightarrow \textit{BOOL}$$

A behaviour-defining equation for this operator might be:

$$\textit{Safe to open?(V, P) = T}\ \text{IF}\ \textit{Pressure in pipe P < Excess and Sensor on V == OK}$$

Animation (direct Specification execution) is performed by treating such equations as left to right rewrite rules. That is, an expression which embodies an instance of a left-hand-side of an equation will have that matching portion rewritten to its corresponding right-hand-side [2, 7].

*Tools:*

Commercially available: ObjEx toolset, Gerrard Software Ltd, UK (UNIX and VAX VMS), [8].

Academically available: OBJ3, Stanford Research Institute, USA, and PRG, Oxford University, UK.

Other tools such as Theorem Support Aids, Test Case Generators etc. are still in their infancy, although these are current research areas.

*Appraisal:*

OBJ is a mature, stable Formal Specification Language with industrially-proven methods of application. It does not demand a vast artillery of Mathematical skills and notational familiarity from its users. As such, it has often been said to be ‘an Engineer’s Formal Specification Language’. OBJ benefits from an easily learnt, and easily understood notation. Benefits both in terms of ease of take-up, and arguably more importantly, ease of specification assimilation and validation.

An OBJ User Group was formed in 1990. (Contact Mr David Nuttall, British Aerospace (Dynamics) Ltd, PB 218, PO Box 19, Six Hills Way, Stevenage, Hertfordshire, UK).

*Maturity:* OBJ has evolved and appeared in various syntactic guises since the mid-1970’s and has a world-wide body of usage experience since that time.

*Availability*: Industrial courses and support consultancy are available. There isn’t a definitive text-book, but an introduction can be found in [7, 10]. A language definition for a commercial tool-supported form of OBJ appears in [9].

*Strength:* OBJ is capable of expressing data-orientated properties and requirements in a simple, easily learnt and readable manner. It is weaker, and requires more effort and user experience, to apply to process-orientated requirements.

*Industrial Experience:* OBJ has been used on a number of projects: GEC-Marconi Secure Systems in the UK have used OBJ in the specification and V&V of trusted, secure systems; Hughes Aircraft Corporation in the USA and AEW, Sweden and Australia have used OBJ in safety-critical applications; Thorn-EMI in the UK have used OBJ in the specification of high-reliability software tools; British Aerospace have used OBJ in systems development trials.

*Tool Availability*: see above

*Applications and Experience Matrix:*

not available

*Tools Matrix:*

not available

*Bibliography*

[1] Richard Shutt, ‘A Rigorous Development Strategy Using the OBJ Specification Language and the MALPAS Program Analysis Tools’, Proceedings of ESEC ‘89, Lecture Notes in Computer Science, Springer Verlag, Eds. C Ghezzi, J A McDermid, pp260-291, 1989.

[2] Joseph Goguen, ‘Parameterised Programming’, IEEE Transactions in Software Engineering, 10 (1984) pp528-543.

[3] Graham Titterington and Christopher Gerrard with guidance from Joseph Goguen ‘Realtime Specification with OBJ’, Gerrard Software Limited, UK, 1992

[4] Joseph Goguen and J Tardo, ‘An Introduction to OBJ a Language for Writing and Testing Formal Algebraic Program Specifications’, Proc.  Conf. on The Specification of Reliable Software, Cambridge Massachusetts, IEEE Computer Society, 1979, pp 170-189.

[5] Christopher Gerrard, Derek Coleman, and Robin Gallimore, ‘Formal Specification and Design Time Testing’, IEEE Transactions in Software Engineering, 1990, SE-16 (1), pp 1-12.

[6] Martin Woodward, ‘Errors in Algebraic Specifications and an Experimental Mutation Testing Tool’, IEE Software Engineering Journal, July 1993.

[7] Christopher Gerrard et al, ‘OBJ - An Introduction’, Gerrard Software Ltd, UK.

[8] ‘DTI Starts Guide’, NCC Publications, UK, 1989.

[9] ‘The OBJ Reference Manual’, Gerrard Software Ltd, UK, 1993

[10] ‘OBJ & Modula2’, R Mitchell, Prentice-Hall, Tony Hoare Series, ISBN 0/13/006081/X, 1992.

## 6 SAGA - LUSTRE

*Summary*

SAGA is a design tool based on the LUSTRE[6] synchronous data flow language. The language can be used for both design and specification. The language has a simple formal semantics which is design to allow the designer to characterise and reason about the behaviour and timing characteristics of systems. System descriptions are intrinsically parallel consisting of a collection of synchronised program components which loop indefinitely to provide the response to stimuli from the operating environment.

*Applications*

The application domain is: reactive systems, real-time embedded systems and real-time automatic control. SAGA has the potential to describe and implement PLC based systems. It is also used for hardware specification and design.

*Properties*

The systems described by LUSTRE are essentially finite-state and thus arbitrary properties of the system may be formally checked using model checking techniques. This feature has been experimented with in an industrial setting but is not yet available on the market. At the moment the SAGA tool provides the means to check:

> *Causality*: In the nth cycle of operation the output only depends on inputs received at or before the nth cycle.
>
> *Bounded Memory:* there must exist a finite bound on the number of past input values necessary to define an output.

*Relation to other formal methods:*

The language belongs to the broad family of process calculi and is closely related to the Meije calculus. The system is intended to provide facilities for the modelling an design of the behaviour of systems. The use of data-flow notation provides a connection to the language LUCID. Other synchronous languages include ESTEREL which has been used in similar application areas.

*Theoretical basis*

SAGA is a, synchronous data flow language. This means that program variables are considered to be infinite sequences of values and a program shows how to generate those sequences. The environment variables are also modelled as infinite sequences which can change to reflect changes in the environment. The successive values can be thought of as the evolution of the system through one tick of a clock. The language is synchronous in the sense that all environmental and programmed stimuli which can happen at the next instant are gathered together and the evolution of

[6] Information on LUSTRE and contact information for tool suppliers can be found at the URL: http://www.imag.fr/VERIMAG/SYNCHRONE/lustre-english.html

the system is calculated to arrive at the new system state. This calculation is considered to be instantaneous.

*Tools*

The SAGA tool has been sold by VERILOG since 1992. Currently the system consists of a compiler plus design tool. This is capable of verifying some basic properties of a system to cheek it is well-formed. Experiments with a tool for formal verification of properties have been carried out inside Merlin-Gerin.

*Appraisal*

SAGA - LUSTRE provides a powerful mechanism for the synchronous modelling of systems based on data-flow notation. The system provides a high level mechanism for the modelling of the behaviour of systems and the data they manipulate. Currently formal verification capabilities of the tools are limited but the formal semantics of LUSTRE means it is possible to construct formal verification tools.

*Maturity:* The method is used in an industrial context for the design of critical software. Merlin-Gerin have used the system in the design of a nuclear power plant emergency shutdown system. See [1, 3] for details of the us of the system. The SAGA tool has been commercially available for the last two years.

*Availability:* The tool is commercially available.

*Strength:* The system provides a powerful means of modelling data and behaviour in systems. Currently, expressing properties of the system is limited but this can be remedied.

*Industrial Experience:* See under maturity. A number of case studies exist, including a study in the area of railway signalling and nuclear protection systems.

*Tool Availability*: Verilog market the SAGA tool. The contact address is:

Daniel Pilaud
VERILOG- Miniparc - zirst rue Lavoisier
38330 Montbonnot Saint-Martin
FRANCE

*Applications and Experience Matrix:*

not available

*Tools Matrix:*

not available

*Further Description:*

SAGA is a data flow language, thus each variable of the program is considered to be an infinite sequence of values and a program sets about defining those sequences. The sequence can be thought of as time progressing in “ticks” of some clock. The following fills in some additional detail on the nature of SAGA.

*Saga is a structured language*

The SAGA language allows the top-down hierarchical decomposition for both data and functions. Controls are made to ensure the consistency of these decomposition with respect to a. strong type concordance.

*Saga is a synchronous language*

Contrary to what we call asynchronous languages in which the time is a “continuous” one (i.e. represented by the set of real numbers), the notion of time in SAGA is a discrete one (i.e. represented by the set of integers).

The synchronous assumption means that theoretically, calculi don’t take any time, no duration is taken into account in the expression of the behaviour. Practically, the time taken by the calculus only needs to be always less than the time required for a new event to occur.

In the case of SAGA, the program scans the inputs, so we can consider that cyclic scanning defines the time scale of the discrete time. Two events are thus seen either at the same cycle (i.e. simultaneously) or at two strictly different cycles.

With the synchronous behaviour, the program calculates synchronously to the scanning period or more practically strictly between two successive scanning actions.

*Saga is a data flow language*

With SAGA, each system, each program is considered as a function which transforms its inputs into outputs. The function implemented by the program is specified as a system of equations which defines outputs from inputs, using possibly some internal variables. The approach of the programming in the usual languages is different from the one in SAGA. Indeed, in the former, defining the system behaviour consists of defining the sequence of actions which leads to the calculus of the outputs from the inputs. In the latter, outputs are defined directly from inputs by means of systems of equations.

Since the systems we implement are “non-terminating”, we will consider data as infinite sequences of values which represent the data evolution all along the system life. The number of the sequence is interpreted as the number of the system cycle. For example, the equation $y = f(x)$ is an equation on sequences and means that the nth term of $y$ $(y_n)$ is defined from the nth term of $x$ $(x_n)$ and equals $f(x_n)$.

In other words, considering that applications are cyclic and successively compute the values of the variables at the first, second,…cycle:

> *$y_n$ is the value at the nth cycle of the application and is calculated from $x_n$ the value of $x$ at the nth cycle of the program.*

The basic operators of the language are the classical ones extended to sequences. As an example, consider the expression

$$X = \textbf{if } Y>0 \textbf{ then} Y \textbf{ else } -Y$$

means

$$\forall i \in \mathrm{N}.X_i = \textbf{if } Y_i > 0 \textbf{ then } Y_i \textbf{ else } -Y_i$$

and defines *X* as the absolute value of *Y*.

There are two specific operators **pre** (standing for previous) and $\rightarrow$ (pronounced "followed by") which are respectively the memorisation operator and the initialisation operator. If *X* is the sequence $\langle X_1, X_2, X_3,\ldots\rangle$ then

$$\textbf{pre}(X) = \langle \textbf{nil } X_1, X_2, X_3,\ldots\rangle$$

where **nil** is the undefined value. If *Y* is the sequence $\langle Y_1, Y_2, Y_3,\ldots\rangle$, then

$$X \rightarrow Y = \langle X_1, Y_2, Y_3,\ldots\rangle$$

This operator is needed to give the variable a particular value at the first cycle of the program.

*Bibliography*

[1] N. HALBWACHS, C. RATEL and P. RAYMOND, Programming and verifying critical systems by means of the synchronous data-flow language LUSTRE, ACM SIGSOFT' 91, conference on software for critical systems, New-Orleans 1991.

[2] P. CASPI, D. PILAUD, N. HALBWACHS and J. PLAICE, LUSTRE a declarative language for programming synchronous systems. 14th Symposium on Principles of Programming Languages, Munich 1987.

[3] J.L. BERGERAND and E. PILAUD, SAGA a Software Environment for dependability automatic control, Proc. Conf. IFAC SAFECOMP88, Fulda 1988.

[4] VERILOG, SAGA user manual

## 7 Z Specification Language - The ZF-set theory and schema calculus

*Summary*

The Z[7] specification language is a model-based approach to formally describing software systems (so-called because one builds an abstract mathematical model of the state and operations of the system to be developed [1]). The major strengths of Z are its powerful, flexible and expressive notation, and its ability to provide structured specifications. Its major weakness is a lack of methodological guidance.

*Applications*

Due to the rich type system of Z, it has been used for a wide range of general computing applications. Specific examples include the specification of file systems, databases, security properties, hardware (i.e. processors), graphics systems, and some work on communication systems.

*Properties*

The strength of Z lies primarily in describing the state of a model, and the operations which can be performed on that state (indeed this approach has led to a number of suggestions for object-oriented extensions to Z, which are summarised in [20]). Both operations and state are typically represented by named schema (a schema is a structuring mechanism which effectively associates variable definitions with a logical formula). The use of a schema typically limits scope of variables used within a particular operation or state to that schema (an axiomatic schema may be defined in which variables have a global scope). Schema may include other schema either as types (in which case the included schema is treated in a similar way as a record definition), or in a form of inheritance. Schema may be defined generically, and instantiated with desired types. Furthermore, it is possible to combine schemas by means of the schema calculus.

Reasoning about Z schemas is done by the conventional proof apparatus associated with first order predicate calculus.

*Relation to other formal methods*

Z is very similar to the VDM technique, particularly since a revised version of VDM provided enhanced modularity. There are differences in syntax (obvious to users) and semantics (hidden from most users). [3] Highlights the main differences.

Process calculus (e.g. CCS, CSP) are very different from Z. Whereas the strengths of the former lie in defining communications between processes, the latter is more suited to a static view of a model.

*Theoretical basis*

Z is based upon a presentation of ZF-set theory in first order predicate calculus. This small core can be extended by means of toolkits (for the most widely known, see [5]) to deal with a, wide range of types (including tupelos, sequences and “enumerated types”).

[7] Information on Z, tools and vendors can be found at the URL: http://www.comlab.ox.ac.uk/archive/z.html

Structuring of Z specifications is achieved by means of the schema calculus, which provides rules for schema to be related by logical operators (and, or, not, implies), or composed in other ways.

*Tools*

A wide variety of tools have been developed for assisting Z specifications, a large proportion of which are summarised in the ZIP Project Z Tools Catalogue [4]. These tools range from public domain through to Commercial. Tools cover many aspects of the development of Z specifications, including: syntax checking, type checking, expanding schemas, proof checking, animation, and calculating preconditions.

*Appraisal*

In addition to its technical strengths, Z benefits from its wide use as a teaching exemplar and the range of application it has found in industrially relevant case studies.

*Maturity:* The Z notation has been in existence since around 1978, and since that time has been gaining considerable popularity in academic and industrial circles. Recently (1990), the ZIP consortium was formed to work towards the definition and development of standards, methods and tools for Z. Deliverables from this group include the Z Base Standard [14], the Z Tools Catalogue [4], and an annotated Z bibliography [15]. Availability: Z benefits from a high profile in both academia and industry.

Courses are offered by a variety of sources (academic and industrial). An increasing number of introductory textbooks are available including [6, 7, 9, 12 and 13]. The Z User Group holds an annual conference, and a large number of papers related to Z frequently appear in the "Formal Methods Europe" conference. An electronic mailing list dedicated to Z related issues is moderated by Jonathan Bowen (requests to subscribe to the list should be sent to zforum-request@-comlab.ox.ac.uk). An archive of Z files (including back issues of the z forum newsletter and a comprehensive Z bibliography) is available via ftp at ftp.comlab.ox.ac.uk or WWW at hhtp://www.comlab.ox.ac.uk/archive/z.html.

*Strength:* Z is particularly suited to expressing specifications for small and medium size applications (although the schema calculus does help with structuring the specification, large specifications will need some additional form of assistance). The type system of Z is sufficiently rich to express a wide variety of data types easily. In its basic form, it is not capable of reasoning about temporal constraints, concurrency and other non-functional requirements.

Comparatively little work has been done on the presentation of refinements (and indeed proofs in general) of Z.

*Industrial Experience:* Z has been used on a wide variety of "real" projects. The Queen's Award for Technological Achievement was awarded in 1992 for the use of Z in the IBM CICS project [16], in 1990 an award was also presented for the development of the IEEE standard for floating point arithmetic on the T-800, which was specified in Z [17, 18]. Craigen et.al.[21] have conducted a survey into industrial usage of formal methods, which covers the above applications, and others besides.

Z has also been used in projects covering the development of secure databases, oscilloscopes, case toolsets, and graphics kernels.

*Tool Availability:* A large number of tools are available for Z, many of which are summarised in the ZIP project Z tools catalogue [4]. The tools are available on a wide variety of platforms. Suppliers include YSE (CADiZ), Logica Cambridge (Formaliser), Mike Spivey, PRG, Oxford University (Fuzz), Bernard Sufrin, PRG, (Zebra) and Xiaping Jia, DePaul University, Chicago (ZTC).

*Applications and Experience Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| **Code** | Work at Durham University is investigating the derivation of formal specifications (in Z and VDM) from “legacy code” [19]. The work performed by IBM and the PRG is aimed at providing a formal description of the interface to the CICS system. Work has also been performed on defining microprocessor instruction sets [22]. | Stepney [2] defines a compiler for high integrity systems. Although the code for the compiler is not given, both the source and target languages are defined in Z (in terms of denotational semantics), and reasoned about formally. | None. |
| **System Models** | A collection of case studies which formalise properties of system models is given in [7], which includes databases, the UNIX filing system, a telephone exchange and an assembler. | Clark et al present a case study of a Motor Speed Control Loop which contains a rigorous proof of the satisfaction of safety properties [10]. Mahoney examines timed refinement in the context of a mine pump case study [8]. | Pilling et. al. [11] provide a model of realtime scheduling algorithms, and give proofs that certain properties are formalised. |
| **Requirements** | Since the modus operandi for Z is to construct a model of the system being built, there is nothing to | See proceeding comment. | See proceeding comment. |

| | | | |
|---|---|---|---|
| | distinguish between requirements and system models (from the perspective of a Z specification). | | |

*Tools Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| **Commercial** | The formalisation of problems generally involves assistance in syntax checking and type checking. The following commercial tools support this: Balzac, CADiZ, fuzz, Formaliser, Genesis. | None | Balzac, CADIZ, Genesis, Mural and ZEDB & B |
| **Non-commercial** | Public domain tools providing syntax and type checking include: Zebra and ZTC. Development tools (not publicly available) include ZADOK, Forsite, IBM Z Tool, ZEE, ZEN, ICL Z tool. | | ICL Z Tool |

*Bibliography*

[1] J. Woodcock, M. Loomes, Software Engineering Mathematics, Pitman, 1989

[2] S. Stepney, High Integrity Compilation - A Case Study, Prentice Hall, 1993

[3] I. J. Hayes, C. B. Jones & J. E. Nicholls, Understanding the differences between VDM and Z, Manchester University, Technical Report UMCS-93-8-1

[4] ZIP Project, Z Tools Catalogue, Document Number: ZIP/BAe/90/020

[5] J. Spivey, Z Reference Manual, Prentice Hall, 1989

[6] A. Diller, Z: An Introduction to Formal Methods, John Wiley and Sons, 1990

[7] I. J. Hayes, Specification Case Studies (2nd Edition), Prentice-Hall International, 1993

[8] B Mahoney, I. Hayes, A Case Study in Timed Refinement: A Mine Pump, IEEE Transactions on Software Engineering, 18(9) September 1992

[9] D. C. Ince, An Introduction to Discrete Mathematics and Formal Specification (2nd Edition), Oxford University Press, 1992

[10] S.J.Clarke, A.C.Coombes, J.A.McDermid, The Analysis of Safety Arguments in the Specification of a Motor Speed Control Loop, University of York Report 136, 1990

[11] M. Pilling, A. Burns, K. Raymond, Formal specifications and proofs of inheritance for real-time scheduling, Software Engineering Journal, September 1990

[12] D. Lightfoot, Formal Specification Using Z, Macmillan, 1991

[13] J. B. Wordsworth, Software Development with Z, Addison-Wesley, 1992

[14] ZIP Project, Z Base Standard, Document Number: ZIP/PRG/92/121

[15] ZIP Project ZIP Annotated Z Bibliography, Document Number: ZIP/Logica/92/103

[16] B. P. Collins, J. E. Nicholls, I. H. Sorensen, Introducing Formal Methods: the CICS experience with Z, in B. Neumann et al., editor, Mathematical Structures for Software Engineering, Oxford University Press, 1991

[17] D. Shepherd, G. Wilson, Making chips that work, New Scientist, 1664:61-64, May 1989.

[18] G Barrett Formal methods applied to a floating-point number system IEEE Transactions on Software Engineering, vol 15 no 5, pp 611-621, 1989

[19] K. H. Bennett, M. P. Ward, Using Formal Transformations for the Reverse Engineering of Safety Critical Systems, pp 204-223, F. Redmill and T. Anderson, editors, Proceedings of the Second Safety Critical Systems Symposium, February 1994

[20] S. Stepney, R. Barden, D. Cooper, A Survey of Object Orientation in Z, Software Engineering Journal, March 1992, 7(2).

[21] D. Craigen, S. Gerhart, T. Ralston, An International Survey of Industrial Applications of Formal Methods, U. S. Department of Commerce, NISTGCR 93/626, 1993

[22] J. Bowen, Formal Specification of the ProCoS/safemos instruction set, Microprocessors and Microsystems, 14(10), December 1990

## 8 RAISE (Rigorous Approach to Industrial Software Engineering)

*Summary*

The RAISE[8] method uses the RAISE Specification Language (RSL) for rigorous software development. RSL is a wide-spectrum language that includes model-oriented features and algebraic ones as well as concurrency constructs. The development method is based on the following principles separate development (of individual modules), stepwise development, invent and verify, and rigorous justification. The method has a defined refinement/implementation relation on which justification of the correctness of development is based. The strengths of RAISE are its powerful specification language, its guidelines for specification and development, and its comprehensive tools. Its major weakness is the low level of automation in the justification tools.

*Applications*

RAISE has been used in a number of different application areas, typically characterised by requirements for high reliability such as aerospace and transportation. Particular examples include train protection systems, satellite image processing, tethered satellite systems, concurrent data servers and car engine control systems.

*Properties*

An RSL specification of a system consists of a collection of modules, some of which may be parameterized schemes that can be instantiated with for example particular types.

The wide-spectrum nature of RSL means that system properties can be expressed in a variety of ways. They can be expressed purely applicatively or imperatively, i.e. by defining a state and operation on it. They can be defined using an algebraic style with sorts and axioms or in a model-oriented fashion with elementary and composite types and operations on values of those types. A system can be defined in a sequential or a concurrent style; the latter using synchronous point-to-point communication over channels.

RSL has a collection of proof rules in addition to its formal semantics. The proof rules are the basis for reasoning about properties of a system. The set of basic proof rules for RSL are given in [19].

*Relations to other formal methods*

The model-oriented parts of RSL are similar to those of VDM, but with some syntactic and semantic differences. The algebraic parts of RSL have been influenced by ACT ONE, Clear and OBJ. The concurrency parts include constructs similar to the explicit style of process definition used in CSP, but with an underlying concurrency model similar to CCS.

RAISE uses the 'invent and verify' approach to development as opposed to the transformational approach which is used by for example PROSPECTRA.

[8] Further information on raise can be found at the URL: http://dream.dai.ed.ac.uk/raise/

*Theoretical basis*

The formal implementation relation of RAISE is based on theory implication, i.e. the theory of an implementing specification must imply the theory of the implemented specification, where the theory of a specification is the logical properties of the specification. This means that RAISE has a proof theoretic approach to implementation.

Properties of a system are described by a collection of RSL modules, each of which may define types, values (functions and constants), variables, channels (for communication), axioms and (nested) modules. Values are either defined by axioms, by pre/post definitions or an explicit expression (a body). The semantics of specified properties of values is loose, i.e. any value satisfying the properties is included in the possible models denoted by a specification.

*Tools*

Currently one set of RAISE tools is available; it includes:

- Syntax-oriented editors with type check for writing and checking the 4 kinds of RAISE entities:
    - modules (i.e. specifications)
    - theories (in which expected properties of modules may be expressed)
    - development relations (in which formal relations between modules may be expressed)
    - hints (in which informal text can be recorded)
- Pretty-printer which generates LATEX source code for RAISE entities.
- Translators from RSL to Ada and C++.
- Justification tools consisting of:
    - Syntax-oriented editor with type check for developing and checking justifications (including formal proofs)
    - Implementation condition generator, which generates necessary and sufficient conditions for the formal implementation relation to hold
    - Confidence condition generator, which generates conditions on expressions based on subtypes, preconditions etc.
    - Simplifier, which automatically discharges simple and trivial justification conditions.

*Appraisal*

*Maturity*: RAISE was originally developed from 1985 to 1990 as part of an ESPRIT project of the same name. During the LaCoS ESPRIT project (1990-1995) it was assessed in a number of industrial trials, and in particular the method and the tools have evolved and been industrialised. The method has been used in combination with existing industry process standards such as the European Space Agency's (ESA's) PSS-05, see [17] and [20].

*Availability:* Both RSL and the RAISE method are described in books published by Prentice Hall, [18] and [19]. Currently RAISE is being taught at a few universities. Tutorials have been given at "Formal Methods Europe" symposia and other conferences. In addition, courses and consultancy are provided by CRI.

On-line information on RAISE is available on the Web system at the following address: http://dream.dai.ed.ac.uk/raise/

*Strength:* RSL is able to express functional properties of a system at different levels of abstraction from a high-level axiomatic one to low-level algorithmic details that are close to a program. Moreover the modularity systems of non-trivial size. Also the inclusion of concurrency constructs in the same language supports a direct expression of parallel systems. RSL does not have built-in constructs for expressing real-time properties. they have to be "modelled" by using other constructs in the language.

*Industrial Experience:* The majority of the industrial experience with RAISE stems from the LaCoS project, where seven European companies used and assessed RAISE. The overall project experiences are documented in [7] and [8]. A number of more detailed technical application descriptions and assessments also exist for the following LaCoS applications:

- "Assessment Report" [5] by Bull describing the application of RAISE on cryptographic protocols.
- "4th Assessment Report" [16] by Inisel based on the development of an image processing system.
- "4th Lloyd's Register Assessment Report" [9] by Lloyd's Register based on a number of smaller applications:
    - a signal condition processing system
    - an engine management system
    - a simulation problem
    - a security model
    - a library of reusable components
- "Report On the Development Process" [1] by Matra Transport describing the application of RAISE on an automatic train protection system.

- "SSI Public Assessment Report on the TCA Application" [6] by SSI based on the development of parts of an air traffic control system.
- "Final Assessment Report" [21] by Technisystems based on the development of a shipping transaction system.

RAISE has been used within an ESA project to formalise central parts of an Instrument Control Unit (ICU), [20].

*Tool Availability:* RAISE is supported by the "RAISE tools" which are commercially available from Computer Resources International (CRI). The tools run on Sun Unix platforms.

*Applications and Experience Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| **Code** | An application by Bull within the LaCoS project formalized an existing concurrent data server with the aim of Ending errors in the implementation. | The Bull application did rigorous justifications of expected properties of the data server specification. | None. |
| **System Models** | Most of the applications in the LaCoS project (see "Industrial Experience" above) involved specifying system properties using RSL. | In [13] safety and progress properties of a concurrent system are demonstrated. [10] specifies and rigorously justifies properties of an industrial robot. | In [14] a formal development of a concurrent system with proof of correctness of the development is presented, |
| **Requirements** | Since a RAISE development typically starts by formalising the requirements and then later constructing refined models, most of the applications in the laces project (see "Industrial Experience" above) involved the formalization of requirements. | None. | None. |

*Tools Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| **Commercial** | The RAISE tools support the syntax and type checking of modules, the recording of development relations and the expression of expected properties of modules in the form of theories. | The justification editor of the RAISE tools support the interactive construction of proofs that may include informal arguments. | The justification editor of the RAISE tools can be used to construct fully formal proofs. |
| **Non-commercial** | None | None | None. |

*Further Description:*

Both [2] and [6] describe projects within the aerospace domain where RAISE has been used and Ada code produced by automatic translation of RSL specifications.

*Bibliography*

[1] M. Abdelaziz. Report on the Development Process. LACOS Report LACOS/MATRA/MA/037/V1, Matra Transport, February 1995.

[2] A. Alapide, M. CineneHa, and P. La Vopa. Automatic Generation of Ada Code with RAISE Formal Method. In *Proceedings of Ada in AEROSPACE '92.* EUROSPACE, 1993.

[3] A. Alapide, M. Cinenella, and P. La Vopa. The Viability of Applying RAISE to Ada Projects in Competitive Bid Companies. In *Proceedings of Ada in AEROSPACE '92.* EUROSPACE, 1993.

[4] D. Bjorner, A. Haxthausen, and K. Havelund. Formal, Model-oriented Software Development Methods - From VDM to ProCoS & from RAISE to LaCoS. Future Generation Computer Systems, (7), 1991/92.

[5] P. Boury. Assessment Report. LACOS Report LACOS/BULL/PB/35/V1, Bull, September 1994.

[6] M. Cinnella, S. Candida, A. Alapide, S. Quaranta, D. Stellacci, and P. La Vopa. SSI Public Assessment Report on the TCA Application. LACOS Report LACOS/SSI/MCSCAADS/1/V3, SSI, February 1995.

[7] B. Dandanell, J. Gørtz, J. Storbank Pedersen, and E. Zierau. Experiences from Applications of RAISE. In *Proceedings of FME '9,1,* volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[8] B. Dandanell, J. Gørtz, J. Storbank Pedersen, and E. Zierau. Experiences from Applications of RAISE, Report 3. LACOS Report LACOS/SYPRO/CONS/24/V1, Computer Resources International A/S, January 1994.

[9] R. Granville (Editor). 4th Lloyd's Assessment Report. LACOS Report LACOSS/LLOYDS/RJG/94/V1, Lloyd's Register of Shipping, January 1995.

[10] F. Erasmy and E. Sekerinski. Stepwise Development of Control Software - A Case study Using RAISE. In *Proceedings of FME '94,* volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[11] C. George. The RAISE Specification Language: A Tutorial. In *Proceedings of VDM* '91, volume 551 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[12] C. George. The NDB Database Specified in the RAISE Specification Language. *Formal Aspects of Computer Science*, (4), 1992.

[13] J. Gørtz. Specifying Safety and Progress Properties with RSL. *In Proceedings of FME '94,* volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[14] A. Haxthausen and C. George. A Concurrency Case Study Using RAISE. In *Proceedings of FME '93,* volume 670 of Lecture *Notes* in Computer Science. Springer-Verlag, 1993.

[15] A. E. Haxthausen, J. Storbank Pedersen, and S. Prehn. RAISE: a Product Supporting Industrial Use of Formal Methods. *Technique et Science Informatiques*, 12(3), 1993.

[16] M. Insausti, R. Codina, and A. Creus. 4th Assessment Report. LACOS Report LACOS/INISEL/TEAM/4/V2, Inisel Espacio, January 1995.

[17] E. Manero. RAISE and ESA Software Lifecycle. In *Proceedings of Ada in AEROSPACE'91.* EIJROSPACE, 1992.

[18] The RAISE Language Group. *The RAISE Specification* Language. BCS Practitioner Series. Prentice Hall International, 1992.

[19] The RAISE Method Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall International, 1995.

[20] P. Taylor. European Space Software Development Environment, Advanced Methods and Tools in EESSDE, Final Report. Technical Report D/2.1.2/E, EDS Ltd, October 1994.

[21] Technisystems LaCoS Team. Final Assessment Report. LACOS Report LACOS/TECH/DR3/6/V1, Technisystems, December 1993**.**

## 9 B[9] method

*Summary*

A full development method from formal specification refined to high-level language, tool support and translation to Ada, C etc. available. The basic idea is of a machine with internal state and operations which are called by other machines or as reaction to environment. State has invariant and operations which can be underspecified and refined later. Similar mathematical foundation to Z underpins notation and language (no schemas but lots of set theory and functions). Highly structured, intial structure is preserved down to code level so it is important for this to be correct.

Could be used without carrying out proofs, i.e. for structuring and code generation.

*Applications*

General purpose but main serious use is on railway signalling systems.

*Properties*

Anything expressible in set-theory, functional behaviour reasoning via set theory and small number of principles and user-defined lemmas.

*Relation to other formal methods*

Foundations similar to Z.

*Theoretical basis*

Set theory, with fixed points, programming and specification constructs defined through extension of substitutions (generalised substitutions).

*Tools*

Atelier-B and B-tool both commercial products.

> *Atelier-B:* is quite fast, robust etc. works under workstations (Sun, HP) and Linux on PCs using X windows, heart of system is a prolog-like core for reasoning. Good project control and information (e.g. regeneration of proofs after changes, tracking of dependencies, information on number of proofs completed/outstanding), uses stand-alone editor on text files (you can change the editor) and could be customised as intermediate files are text based. Theorem proving now combines automatic with interactive proof for lemmas. Now feasible to prove developments completely by adding required lemmas and proving them.
>
> *B-tool:* provides similar facilities also under X windows on workstations (availability for PCs not known), includes animation of initial spec (under development for Atelier B). Perhaps not as fast and theorem prover rules less comprehensive.

[9] Further information on the B method can be found at the URL: http://www-scm.tees.ac.uk/bresource/docs/B/BROOT.html or at: http://www.atelierb.societe.com/PAGE_B/uk/atb-01.htm

*Appraisal*

*Maturity:* Yes, role in process is developing e.g. in REAIMS esprit project.

*Availability:* Medium, improving as time passes. Books now available (see bibliography), training available now through B core, and from Steria-Mediterranee for Atelier B.

*Strength:* Good integrated development method, specs perhaps a little limited in style, much development effort put in e.g. by French institutions and BP (original effort).

*Industrial Experience:* GEC Alsthom use Atelier B on 10+ industrial projects for real safety-critical and safety-related code. Matra transport use Atelier B to develop metro line control system. B-tool used in trial with Lloyds Register and in GEC Marconi aerospace.

*Tool Availability:* Good, B-tool from B-core and Atelier B from Steria-Mediterranee France.

*Applications and Experience Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| **Code** | Not applicable | Not applicable | Yes, GEC Alsthom, Matra |
| **System Models** | No | No | Yes, GEC Alsthom, Matra |
| **Requirements** | No | No | Yes, GEC Alsthom and REAIMS |

*Tools Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| **Commercial** | | | Atelier B, B Tool |
| **Non-commercial** | | | |

*References*

[1] J. R. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996

[2] B. Dehbonei, F. Mejia, *Formnal Development of Safety-crtitical Software in Railway Signalling*, in Application of Formal Methods, ed. M G Hinchey and J P Bowen, Prentice Hall 1995

[3] Proceedings of 1st Conference on the B method (Nantes, 25–27 Nov. 1996) (available from IRIN, University de Nantes-Ecole Centrale de Nantes)

[4] John Wordsworth, *Software Engineering with B*, Addison Wesley Longman, 1996

[5] Kevin Lano, *The B Language and Method: A Guide to Practical Formal Development*, Springer-Verlag, FACIT series, 1996

## 10 VSE - Verification Support Environment

*Summary*

VSE[10] is a tool that permits formal software development, from formal specification and verification to (partly) automatic code generation, including support of the development documentation. Within the formal specification the security (safety) model, the formal top level specification and refinement steps are created with the formal specification language VSE-SL. The deductive component of the VSE system for automatic proof obligation generation and semiautomatic proofs consists of integrated versions of KIV (Karlsruhe Interactive Verifier) and INKA (Induction prover Karlsruhe). The KIV system is mainly used to handle proof obligations involving programs, while INKA is used for proving first-order predicate logical subtasks (e.g. verification conditions) including induction.

*Applications*

VSE is a general purpose tool for formal development of sequential critical systems within the security and safety area.

*Properties*

Properties are described in security- or safety models within the specification language VSE-SL. The dependencies between functional specification and these models are defined in SATISFIES-relations that have to be formally proved. Because of transitivity the dependencies hold through all following refinement steps

*Relation to other formal methods*

Both the method and main characteristics of the tool can be somehow related to the B-Method and B-Tool.

*Theoretical basis*

The specification method is based on abstract data types and state transition systems using first order predicate logic (with equality). The proof component consists of two theorem provers (KIV, INKA).

*Tools*

The VSE-Tool is commercially available. A list of distributors can be requested through BSI, Ref. II1, Attn. Mr. Koob, Postfach 20 03 63, 53133 Bonn, Germany

*Appraisal*

*Maturity*: The tool is based on a method that had been developed in universities and research institutions for the last 10 years. It combines that well founded method with up to date development techniques including interfaces to conventional CASE-tools.

[10] Further information on VSE can be found at the URL http://www.dfki.uni-sb.de/vse/projects/

*Availability* Consultancy, textbooks and introductions, training, courses etc. are available through the a.m. address.

*Strength:* Being a general purpose tool for formal software development it naturally has weaknesses if the method and the tool is compared to specialized methods. Of course in general model checking works better where it can be applied.

*Industrial Experience:* Two case studies with large systems exists. After the development of VSE eight pilot projects in the areas of both security and safety were carried out. In 1995 VSE was officially recommended for formal software development and evaluation according to ITSEC (Information Technology Security Evaluation Criteria). Currently VSE is used in two real industrial system development projects and three ITSEC evaluations.

*Applications and Experience Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| Code | | | |
| System Models | | | |
| Requirements | | | |

*Tools Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| Code | | | |
| System Models | | | |
| Requirements | | | |

*Further Description*

Currently, 1996-1999, VSE is modified and improved in a follow on project. Based on Temporary Logic of Actions (TLA), VSE-II will be able to handle specification and correctness proofs of concurrent and real-time systems. Further on besides optimisation of the deduction component interfaces to Model Checking and the Z language will be implemented

*Bibliography*

[1] F. Koob, M. Ullmann, S. Wittmann, Industrial Usage of Formal Development Methods - the VSE-Tool Applied in Pilot Projects, In: Proceedings of COMPASS ‘96, Gaithersburg, MD, USA 1996, ISBN 0-7803-3390-X

[2] D. Hutter et al, Deduction in the Verification Support Environment (VSE), In: Lecture Notes in Computer Science 1051, Proceedings of FME ‘96, Oxford, UK, 1996, Springer 1996

## Appendix A Filling out a new entry

Each entry in the guideline has a standard format which should help the potential user assess the suitability of the method for their application. Each entry in the directory has been generated by individuals familiar with the particular method. The format and the instructions supplied to those completing an entry are:

*Method:*

Give the title of the method.

*Summary:*

This should provide a brief summary of the method, outlining the main areas of application, its strengths and weaknesses.

*Applications:*

Summarise the main areas of application.

*Properties:*

Discuss the kinds of properties which can be described by the method and the ways of reasoning about them.

*Relation to other formal methods:*

Point out closely related methods and contrast with radically different methods. The main aim is to try to give some cues to potential users.

*Theoretical basis:*

Try to be fairly non-technical - describe the kind of computational model an how one goes about specifying properties of computations.

*Tools:*

Give a summary of available tools, their function, robustness and scope. Identify their strengths and weaknesses

*Appraisal:*

Give a brief introduction to the five appraisal points given below.

> *Maturity*: Assess how technically mature the method is and its maturity in terms of process, i.e. harmonising with other software development techniques.
>
> *Availability*: Assess the availability of consultancy, textbooks and introductions, training, courses etc.
>
> *Strength:* Assess the kinds of areas the methods is strong in expressing requirements/properties and where it is weak.

*Industrial Experience:* Try to be as concrete as possible, preferably point to specific projects and companies which have experience of the method. If there are case studies in the open literature then point to these too. Tool Availability: Assess the availability of tools. Give names of suppliers in the case of commercial tools or institutions developing systems otherwise. There is no need to be comprehensive here

*Applications and Experience Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| Code | | | |
| System Models | | | |
| Requirements | | | |

*Tools Matrix:*

| | **Formalization** | **Rigorous Proof** | **Formal Proof** |
|---|---|---|---|
| Code | | | |
| System Models | | | |
| Requirements | | | |

For each cell point to some work which has been done in the area. Try to give references where possible. If there is no work in the area then say so.

## Appendix B  Classification of methods by their theoretical basis

In a perfect world we could give a complete characterisation of the various qualities of evidence we could expect from each different formal method but unfortunately at the moment this is not possible.  Ultimately the qualities of evidence generated by the application of any particular method depend on the development of a *body of practice* with the method.  Use in practice is the best indicator we have of the strengths and weaknesses of particular methods.  Unfortunately we are only at the beginning of our experience with the application of formal methods thus this section is rather sketchy.  This classification thus takes the theoretical basis of the method as its basis.  This will change as we gather more evidence of the success of formal methods in tackling industrial case studies.

At the coarsest level of characterisation,  it is possible to divide formal methods into two broad categories: those which centre on properties of data in the system and those which centre on the interaction.  Data centred methods are well suited to modelling algorithms and data manipulations where we want to focus on correctness of some transformation of inputs to outputs.  Interaction centred methods focus on the interaction a system can undertake and so focus on showing properties like some action will never happen or some action will always eventually happen.  Of course when we analyse any system we are usually concerned with both kinds of property.  This suggests we might have to use two methods.  However most formal methods have some mechanisms for dealing with both kinds of property.

### *Data centred methods*

These methods concentrate on specifying functions or operations to be carried out in the operation. A variety of specification methods are advocated.  The algebraic approach is usually associated with *axiomatic*  specification where the specifier is called on to define the “laws” which the operations will obey.  The other approach is to *model* the operations in some pre-existing structure.

#### *Algebraic methods*

This class of formal methods grew out of work on algebraic specification.  The basis of the work is that certain classes of theories specify algebras and algebras are the natural mathematical model of systems.  By providing the means to structure theories and to explore the relationships between different structured theories one can provide a fully formal system within which system development can proceed.  As it is presented in the literature this approach tends to look like “all or nothing” but we do not believe this needs to be the case selective application of the algebraic approach can be beneficial both at the requirements stage when the developed theories can be useful in characterising the data in the system.  In design and coding the notions of refinement and coding contained in these approaches may be very helpful.  It may also be that the algebraic characterisation of the system is helpful in determining the adequacy of test suites.

Historically the system of Bauer  is the first algebraic specification language but since then a number of other systems have been developed, for example: Extended ML, Act One (Two), Larch[11], OBJ (n) . These are the some of the most recent methods taking this approach many others have led to this stage of development. The Compass[12] Esprit BRA attempted a harmonisation of approaches this has been quite fruitful leading to the follow on Common Framework Initiative (CoFI[13]). An important development in these systems has been the move away from strict mathematical equality between objects to a focus on observable behaviour of algebras. Most modern methods in this class take this approach.

Tool support has always been a strong part of this approach and almost all the methods mentioned here offer some form of support from typechecking structured specifications to offering interfaces to general purpose proof assistants to support the refinement process.

*Logics and type theories*

These methods are generally based around some kind of logic in which the objects considered are classified into some type system and some of those objects can be considered to be programs. The commonest kind of computational object is the higher- order function (such systems are oriented towards functional programming languages) though recent work has attempted to take concurrent processes as objects. There is a great deal of work in the area of tool support for this kind of work, much of it falling under the ESPRIT BRA projects: Types and Proofs for Programs and CLICS (Categorical Logic in Computer Science)[14]. Broadly one can categorise by the logical system or systems underlying the tool. Roughly we have:

*Martin-Löf Type Theory:* This is a particular version of type theory developed by a Swedish logician. It is particularly well-suited to the specification and development of programs in a fully verified manner but as with all the systems mentioned below it has substantial power to model systems and can provide useful information at stages other than design and development. Notable amongst the tools supporting this theory are: ALF and NuPRL[15]. These systems are full proof assistants for very powerful logics substantial experience is available in the area of general theorem

[11] The Frequently Asked Questions list for Larch provides an excellent source of information on Larch and a variety of other formal methods. You can find it at: http://www.cs.iastate.edu/~leavens/larch-faq.html

[12] A summary of the results of Compass can be found on the home page of the project, see: http://www.informatik.uni-bremen.de/~inform/forschung/compass/compass.htm. This has a summary and extensive bibliography.

[13] CoFI has a home page at: http://www.daimi.aau.dk/~pdm/Common This project is attempting to integrate the approaches of ASF+SDF, COLD, Extended ML, Larch, OBJ 3, RAISE, Spectrum and Z. The home page of the project is a convenient starting point to reach more information on all these approaches.

[14] The Types basic research action has a home page at: http://www.dcs.ed.ac.uk/lfcsinfo/research/types_bra/index.html, there one can find an index of the systems developed under the BRA.

[15] The NuPRL home page is at: http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html

proving and some for system development. NuPRL has been used for a number of industry-related applications including the development of circuits.

*Calculus of Constructions:* This system is similar to type theory in its expressiveness various systems implement this and its extensions. Notable here are Coq[16] and Lego[17]. Work has been done on aspects of system development using these systems. Recent work using Coq has centred on the use of the system in the specification and development of safety-critical systems.

*Higher Order Logic:* These systems owe their origins to Scott's logic LCF and have the longest pedigree in work supporting system development. In particular there has been substantial work done on the development of synchronous hardware via a method usually called *regimentation* where time is discretised to the ticks of a clock. Notable tools supporting such logics are: HOL[18], Isabelle[19] and Lambda[20]. For HOL and Lambda, there is good support for hardware design via pre-specified components and special purpose tools and libraries which ease the routine tasks in the development process.

*Other Logics*: Most notable here is the nqthm/acl2[21] logic developed by Boyer and Moore and supported by Computational Logic Inc. There is extensive experience of the use of the system to support the development of computer systems. The method is focussed on the automated support tool for the logic. More recently the PVS[22] system has been applied to a number of industrially related topics including circuit development and avionics systems.

*Specification/Program Logics*

Broadly one can see these methods as having their intellectual history in the insights into programming gained by Hoare in the 70's. The methods most commonly encountered in this category are Z[23] and VDM[24]. Both approaches take the provision of a collection of sound

[16] Details of Project Coq which is exploring the use of Coq in the development of zero-defect software can be found at: http://www.inria.fr/Equipes/COQ-eng.html

[17] The Lego home page is: http://www.dcs.ed.ac.uk/packages/lego

[18] The home page for HOL can be found at: http://www.cl.cam.ac.uk/Research/HVG/HOL/index.html

[19] The home page for Isabelle can be found at: http://www.cl.cam.ac.uk/Research/HVG/isabelle.html

[20] For information contact Abstract Hardware Ltd at lambda@ahl.co.uk.

[21] Details of nqthm/acl2 can be found at the CLINC web page: http://www.cli.com/

[22] See http://www.csl.sri.com/pvs.html for more information. The work on PVS is funded by NASA and the work surrounding PVS takes avionics as an important (though not exclusive) focus.

[23] See the entry in this document for more details.

[24] The main homepage for VDM tools can be found at: http://hermes.ifad.dk/products/vdmslproducts.html

mathematical modelling tools as the basis of their approach. Both are based on classical set theory but there are differences in the precise formulation of the mathematical foundations. The approach to specification is to provide a mathematical model of the system to be defined and then to provide specifications of the operations which bring about state change in the system.

The expressive set theories provide a convenient way of describing the system state while abstracting away from unnecessary detail. The operations are described by giving pre-and post-conditions on the state. These conditions are given in a logical language. The precondition expresses the requirement on the state for a successful application of the operation and the post-condition describes the change of state which should be wrought by the operation.

The design of these formal methods is carefully matched to the needs of the development of systems which consist of a collection of operations which bring about changes of state. Both methods have fairly well developed tools available but so far few are of industrial standard. The style of tool ranges from type-checkers which provide fairly basic cheek on the syntax of the specification to systems which provide full theorem proving capabilities.

*VDM*: VDM has a well developed system of refinement intended to demonstrate that an implementation is a refinement of a, specification. There is a standard for VDM which does provide for some modularisation of the presentation of specifications. There is an extensive literature on the application of VDM to industrial style applications. A number of institutions run courses on the use of VDM.

*Z*: As it stands Z is exclusively a specification method. Its most distinctive feature is the “schema calculus” which provides the means to structure specifications at a high level by providing a variety of composition methods, parameterised schemas etc. Though Z is only a specification method, a refinement calculus has been developed as part of the B-tools effort (see the entry for the B method in this directory). This method provides the means to derive implementations from Z specifications. There is an extensive literature on the application of Z to industrial style applications. A number of institutions run courses on the use of Z.

## *Interaction centred methods*

Data centred methods have some built-in notion of the kinds of data one expects to use during a computation - but the do not have a model of how a system interacts with the outside world built-in as an intrinsic part of the method. Methods centred on interaction take such a model of interaction as fundamental and structure the method around that

### *Automata based methods*

These methods use some notion of automata as the fundamental model. Thus one defines the collection of sates and the transitions which can be made by the system. Statecharts and its support tool Statemate[25] is one method with fairly wide industrial use. This provides a structured way to

[25] See the URL: http://www.ilogix.com

design finite automata to describe the dynamics of a system. The support tool provides an environment to support graphical presentations of the system.

Another widely used automata based approach is Petri Nets[26]. These are used extensively in the specification and verification of communication protocols and in a number of other areas. The industrial applications of Petri nets is quite mature

*Process calculi*

Process Calculi are useful in developing abstract models of processes during the design phase and can support a notion of refinement and implementation. Usually one can reason about such systems using modal or temporal logic. Automated tools such as the Edinburgh Concurrency Workbench coupled with judicious abstraction can support a rather exploratory approach to designing systems while maintaining certain important properties. The properties to be checked are usually expressed in some kind of modal or temporal logic. This is a very active area of work and a number of parallel lines of enquiry are being pursued. Broadly the area can be subdivided between asynchronous calculi which assume no global synchronisation and synchronous calculi which assume some king of global synchronisation. The CONCUR[27] projects focus mostly on asynchronous calculi and their developments into synchronous calculi. The other approach to be tested in an industrial context is the use of synchronous programming languages as design tools. See the SAGA-LUSTRE entry in this directory.

*Model checkers*

This is a very active area of industrial work. These techniques aim to provide automated support for the symbolic debugging of interactive systems. A large number of industrial organisations are experimenting with and implementing the use of model checkers in their design process. A representative of this approach is the PROMELA/SPIN[28] system at Bell Labs. This provides the means to specify systems and check they have specified interaction properties.

### *Formally defined programming languages*

One problem with applying any method for reasoning about systems is a lack of rigour in defining the meaning of the programming language in use. Recently the description of programming languages has become substantially more rigorous. Examples of this are:

Standard ML[29]—the description of the language is published and is frozen. The language incorporates a very expressive type system incorporating a feature called polymorphism which

[26] There is an extensive literature on Petri nets and a large number of support tools. One good starting point is: http://www.daimi.aau.dk/~petrinet/

[27] See the URL: http://www.sics.se/fdt/concur2.html

[28] See the URL: http://netlib.bell-labs.com/netlib/spin/whatispin.html

[29] See the URL: http://foxnet.cs.cmu.edu/

encourages software reuse.  The language is capable of supporting both functional and imperative programming and (via the system of modular programming incorporated in the definition) limited features of object oriented programming.

SPARK Ada[30] - this is a subset of Ada with a rigorously defined semantics and industrial strength reasoning support.

These languages provide a, rigorous operational semantics which provides an unambiguous standard against which implementations can be measured and provides the basis for techniques for modelling and reasoning about program behaviour.  Often tool support is reasonably good.  For example:

*ML type inference:* This generates timely and accurate information during the process of coding.  It is very useful for internal documentation.  Takes a little while for programmers to get used to and is completely unintelligible to a user/customer.  It is inadequate to describe the functionality of a system.

[30] See the URL: http://www.praxis.co.uk/service.lines/critical.systems/spark/sp_lang.stm