Discrete Mathematics, Formal Methods, the Z Schema and the Software Life Cycle

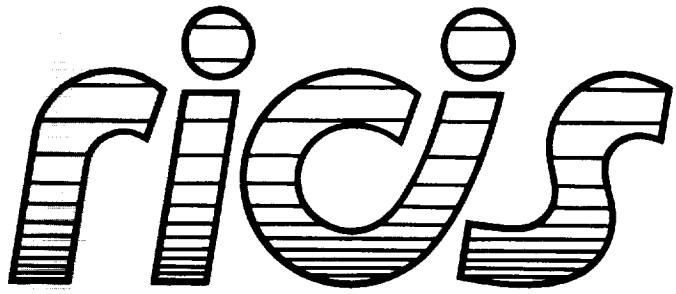
Rodney L. Bown

University of Houston-Clear Lake

April, 1991

Cooperative Agreement NCC 9-16
Research Activity No. SE.26

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Discrete Mathematics, Formal Methods, the Z Schema and the Software Life Cycle

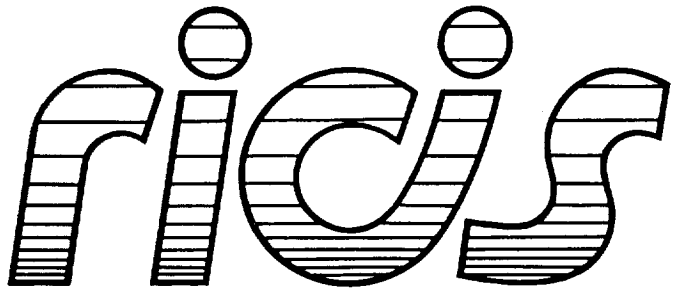
Rodney L. Bown

University of Houston-Clear Lake

April, 1991

Cooperative Agreement NCC 9-16
Research Activity No. SE.26

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Rodney L. Bown, Associate Professor of Computer Systems Design at the University of Houston-Clear Lake. Dr. Bown also served as RICIS research coordinator.

Funding has been provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was William C. Young, of the Project Integration Office, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the author and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

RICIS TECHNICAL REPORT

Discrete Mathematics, Formal Methods, the Z Schema and the Software Life Cycle

Principal Investigator

Rodney L. Bown

in partial fulfillment of

RICIS Task SE.26

April 1991

CONTENTS

Discrete Mathematics, Formal Methods, the Z Schema and the Software Life Cycle.

This paper is submitted in partial fulfilment of RICIS Task SE.26. The author and principal investigator is Dr. Rod Bown, University of Houston - Clear Lake.

I. Introduction

This paper discusses the proper role and scope for the use of discrete mathematics and formal methods to in support of re-engineering of security and integrity components within deployed computer systems. It is proposed that the Z (pronounced "Zed") schema can be used as the specification language to capture the precise definition of system and component interfaces. This can be accomplished an object oriented development paradigm. One such effort is presented in [WOOD90].

II. Requirements

The software engineering life cycle starts with a set of requirements written in precise language using the terminology of the customer's application domain. The requirements document serves as the legal technical interface between the customer and vendor. A third party judge should be able to use the requirements document to measure success or failure of the delivered system. For this reason a requirement is a statement of intent written in such a way that a metric can be used to measure success or failure of the developed system.

Requirements are captured by system modeling techniques which include viewpoint analysis and check lists. Controlled Requirements Expression (CORE) is one methodology that supports the gathering and analysis of requirements. Requirements can be supported by diagrams and tables that are understood by the customer and a potential third party judge. The customer should be able to understand all requirements without special training.

In a sense, the requirements document is the specification for the set of tests that will be performed to validate the product. The requirements document may contain a glossary that provides precise definitions for all unique terms related to the application domain.

Each design module or activity will be a response to a requirement. In accordance with a specified standard, numerical labels are attached to each requirement and all resulting design activities. This will allow a software management tool to automatically provide forward and backward audit trails.

A rationale subparagraph should be contained in each traceable requirement. This will provide guidance during the design, maintenance and enhancement phases of the life cycle. The rationale serves two purposes. One purpose is to provide the context in which the requirement was stated in order to restrict undesirable side effects that could result from modifications to implemented modules. The second purpose is to act as a forcing function for future modifications. If the rationale has changed, the implementation may need modification.

There should be a limited amount of statements related to the implementation issues of the delivered system. Requirements that constrain the design alternatives are usually related to industrial standards for hardware and software interfaces.

This discussion has been provided to substantiate the argument that requirements will continue to be written in precise natural language such as English. There are two customers (readers) for the requirements document: the customer and the design team. It is the design team that will transform the precise English, diagrams, and tables into a formal specification document.

III. Specifications

The design team is responsible to transform the external requirements document into an internal system specification document. The specification document should use precise terminology that is based on mathematical, physical, and computer domains. The terminology of this document need not be understood by unsophisticated readers. An audit trail is supported by an assertion that a particular specification is the formal presentation of a requirement.

As a special note, different terms appear in the literature. One author uses the terminology of requirements definition and requirements specification [SOMM89]. The requirements definition is a natural language description of the requirements. The requirements specification is a more formal requirements description.

IV. Formal Methods

Computer systems that control life and property are becoming common place within society. History indicates that when a technology enters common usage, society will demand assurance that the design was completed in accordance with accepted mathematical and physical models and good design practices supported by standards. In a like manner software customers (society) will insist that the design of software systems must be based on mathematical and physical models that are supported by some level of formal proof and/or history of proven acceptance. This will include the use of appropriate standards. During a

1989 visit to UHCL, Dr. John McHugh presented this point in an elegant discussion of the software engineering failure that resulted in the Internet worm incident [McHU89].

Formal or precise methods are becoming a requirement for the development of safety critical software. The term safety critical includes the security and integrity issues of software systems. Woodcock and Loomes in their book define formal methods as the class of formal systems which have been designed to be useful specially for the development of complex systems, together with the associated manuals and courses which give rise to guidelines for the formal system's use on real problems [WOOD89]. No one formal system is ever likely to be suitable for describing and analyzing ***all*** aspects of a complex system. A formal method is not ***the*** method which a system designer might choose to use when developing a system, but ***one of the tools*** that a designer might wish to make use of during the process.

V. Z Schema and a Small Example

The following discussion has been extracted from [WOOD89] and modified for this document. In a specification, one can see a pattern occurring over and over again: a piece of mathematical ***structure*** which describes some constrained variables. The introduction of variables under some constraint is called a ***schema***. One schema that shows promise for the specification of Ada programs is Z (pronounced "Zed").

This section will describe part of a configuration manager that maintains modules written in a language such as Ada. Modules have names drawn from the set ***Name***. For this discussion regard the parameter ***Name*** as a parameter of the specification. Such a parameter is called a ***given set***, and is regarded as a primitive type. There is another given set ***Body*** of all (syntactically correct) module bodies.

A module may import definitions (of program functions, procedure, types, etc.) from other modules. For this example consider three program modules. The module ***Application*** needs to write a string of characters to the screen. Since this is such a common thing to do, there is a standard module that deals with "transput" and contains the definition of ***writestring***. ***Application*** imports the definition from ***Transput***. ***Transput*** in turn implements ***writestring*** in terms of another lower-level routine called ***putchar***, which is imported from the first module, ***ScreenHandler***. The reader should see the analog to the predefined TEXT_IO package in Ada.

At this level of abstraction the text of a module consists of a triple: the name of the module; a set of names of imported

modules; and the body of the module. A module should not import itself.

$$\frac{Text : \mathbb{P}(Name \times (\mathbb{P}\, Name) \times Body)}{Text = \{n : Name; s : \mathbb{P}\, Name; p : Body \mid n \notin s \bullet (n, s, p)\}}$$

The three concrete examples of Text are

```
t1 = (ScreenHandler, {},
      module ScreenHandler
        begin proc putchar ... end)

t2 = (Transput, {ScreenHandler},
      module Transput with ScreenHandler
        begin proc writestring ... putchar ... end)

t3 = (Application, {Transput},
      module Application with Transput
        begin proc ... writestring ... end)
```

This can be seen in fragment form as

```
t1    module ScreenHandler
      begin
        proc putchar
        ...
      end

t2    module Transput with ScreenHandler
      begin
        proc writestring
        ... putchar ...
      end

t3    module Application with Transput
      begin
        ... writestring ...
      end
```

These relations can be described using discrete mathematics. For this example a text imports those modules named in its set of imports:

$$\frac{\_imports\_ : Text \rightarrowtail Name}{(n_1, s, p)\ imports\ n_2 \Leftrightarrow n_2 \in s}$$

The program fragments provide the following facts about imports:

$$t_1 \notin dom\ imports$$
$$t_2\ imports\ ScreenHandler$$
$$t_3\ imports\ Transput$$

Another relationship can be built on ***imports***, but it is a relation on ***Texts***: one text ***needs*** another if it imports the second's name:

$$\_needs\_ : Text \leftrightarrow Text$$
$$(n_1, s_1, p_1)\ needs\ (n_2, s_2, p_2) \Leftrightarrow (n_1, s_1, p_1)\ imports\ n_2$$

The set of module texts can be divided into "compatible" modules. One can regard the name of a module and the set of module names that it imports as constituting the "signature" of a module. Two modules are compatible if they share the same signature:

$$\_compat\_ : Text \leftrightarrow Text$$
$$(n_1, s_1, p_1)\ compat\ (n_2, s_2, p_2) \Leftrightarrow n_1 = n_2 \wedge s_1 = s_2$$

The specification of the configuration manager is continued by describing its ***state***. The state consists of those data structures that are maintained by the system; the operations in the system manipulate these data structures and therefore change the state. Mathematical data types are used to build a theory of the state which will have as models suitable realizations in a programming language. This example will show how the state is initialized and how the values of the state before and after operations are related.

The state contains two components: a ***store*** of ***Texts*** and a relation which describes when one stored text is a version which is a ***successor*** for another stored text:

$$store : \mathbb{P}\ Text$$
$$\_suc\_ : Text \leftrightarrow Text$$

The successor can be constrained in the following ways. A module must never be the successor of itself, for that would introduce an unbreakable "loop" when searching for the end of the chain. Thus the ***transitive closure*** of ***suc*** is irreflexive:

$$suc^1 \in Irreflexive[Text]$$

where ***Irreflexive***[X] is the set of all irreflexive relations on X:

$$Irreflexive[X] \mathrel{\widehat{=}} \{R : X \rightarrowtail X \mid (\neg \exists x : X \bullet x\, R\, x)\}$$

If text t_1 is the successor of another t_2, then they must be compatible, that is ***suc*** is merely a subset of ***compat***:

$$suc \subseteq compat$$

All the nominated modules that have, or are themselves, successors must reside in the store:

$$suc \subseteq store \times store$$

store and ***suc*** constitute a proper state of the configuration manager whenever they satisfy these three constraints:

$$suc^{t} \in Irreflexive[Text]$$
$$suc \subseteq compat$$
$$suc \subseteq store \times store$$

Discrete mathematics is sufficiently powerful to describe many aspects of software systems. However, its application to large scale specifications soon results in unwieldy descriptions that are difficult to follow. It isn't the language that is at fault, but rather the human need to comprehend just a small amount of information at a time. One of the most basic things that can be done to improve a specification is to identify and name commonly used concepts and factor them out from the rest of the description of a system. Comments on mathematical notation are presented in the Appendix.

In specifications, a pattern occurs over and over again. This is a piece of mathematical ***structure*** which describes some constrained variables. The introduction of variables under some constraint is called a ***schema***. The Z schema uses a template that provides for the schema, a ***declaration*** of some variables, and a ***predicate*** constraining their values.

Name

declarationpart
predicate

When there is a change in state, the convention is to "decorate" the names of the "after" variable with a dash.

The configuration manager has three predicates that characterize its state, which together form ***the state invariant***. The state invariant, together with the declarations of ***store*** and ***suc*** form a schema which is called ***ConfigMan***. This package of definitions is a mathematical structure that describes all legal values of the state of the configuration manager:

ConfigMan

$$store : \mathbb{P}\, Text$$
$$\_suc\_ : Text \leftrightarrow Text$$

$$suc^{+} \in Irreflexive[Text]$$
$$suc \subseteq compat$$
$$suc \subseteq store \times store$$

This schema is ***equivalent*** to that previously presented. Each line of the predicate forms a conjunct with the other lines.

The initial state of the configuration manager is an empty store and an empty successor relation. The initialization of a system can be regarded as a peculiar kind of operation that creates a state out of nothing; there in no before state, simply an after state, with its variables decorated with a '. The decoration ' is used for labelling the final state of an operation.

InitConfigMan

$$store' : \mathbb{P}\, Text$$
$$\_suc'\_ : Text \leftrightarrow Text$$

$$store' = \{\}$$
$$suc' = \{\}$$

There is only one state of the configuration manager that satisfies this description. There must be at least one, otherwise we could not implement the system.

If the name of the schema is decorated with ', it is understood to mean the same mathematical structure, but with component names

so decorated. For the configuration manager, the state after an operation is described by ***ConfigMan'***. In detail this means:

$$
\begin{array}{|l}
\textit{ConfigMan}' \\
\hline
\_suc'\_ : \textit{Text} \rightarrow \textit{Text} \\
store' : \mathbb{P}\ \textit{Text} \\
\hline
suc' \subseteq store' \times store' \\
suc' \subseteq compat \\
suc'^{t} \in \textit{Irreflexive}[\textit{Text}] \\
\hline
\end{array}
$$

The notation introduced has increased our ability to capture specification of software systems within a ***schema***. The Z schema provides for schema to include previous schemes. This can be seen in the simple example of a hopper which has been extracted from [SOMM89]. The specification of a Hopper includes the schemas for Container and Indicator in its declarative part. The expanded specification for the Hopper does not use the previous schemas. The reader can see that the expanded specification includes all of the details which clutters the presentation. Additional schemas are included that exhibit the filling operation of the hopper.

In this paper the discrete mathematics and Z templates have been extracted from the cited references. This writer assumes that Z schema editors and proof tools will be available in the foreseeable future. At the present time, it is not worth the effort to use Word Perfect to create the mathematics and templates.

VI. Case Studies

The September 1990 issue of the IEEE Software magazine contained two articles that presented discussions of successful applications of the Z schema. The design of an X-ray machine was discussed by [SPIV90]. The design of an oscilloscope was presented by [DELI90].

Of special interest to NASA is the case study discussed in [JACK90]. Jacky has reported on the Clinical Cyclotron Therapy System at the University of Washington. This is a cyclotron and radiation therapy facility that provides cancer treatments with fast neutrons, production of medical isotopes, and physics experiments. The control system handles over one thousand input and output signals.

The designs are attempting to achieve high reliability and safety by applying rigorous software development and quality control practices. They have been intrigued by the possibility that

formal software development techniques might result in additional improvements in reliability and safety beyond those achieved through traditional practices, namely, English like specifications, subjective design and code reviews, and lots of testing.

Other important design goals are improved maintainability and adaptability to future hardware and software modification. This appears to be a direct analogy to space shuttle enhancement and space station design goals.

The project is apparently the first application of formal methods to an accelerator control system. In the literature a few reports describe acceptance testing from the customer's point of view; none discuss design internals nor development practices. A typical presentation emphasizes hardware organization. Discussion of software is limited to informal treatment. There is little interest in development methodology; the very concept of "specification" is practically absent from this literature.

Jacky states that the best known natations are Z and VDM but that Z appears to be gathering more published tutorials and case studies. The author has observed that the distinguishing feature of Z is the *schema calculus*, which provides a convenient way to build up large specifications from textually separate components called *schemas*. In Z, specifications consist of two principle components: descriptions of abstract data structures that model the internal state, and definitions of operations that manipulate the state. For NASA this is the key observation. The schemas provide a precise modeling technique to support object oriented design of the software.

The author is not an experienced Z user. He has provided samples that are intended to show how some of the control system requirements might be expressed in the Z style, and to reveal some of the difficulties encountered by self-taught practitioners.

The article provides Z schemata examples listed below:

- Names and definitions for control parameters.
- Units and conversion formulae.
- Example of schema inclusion. - MACROS
- Constraints on values.

The schemata cited above provide documentation for the database.

Z schemata are shown for display (Xi schema - values are unchanged)

- Delta schema, operation changes the state.
- A software interlock.
- Error messages
- timing constraints.

There are difficulties using Z to represent interdependent collections of serial and parallel operations, event-driven operations, and concurrency. Note: the LOTOS technique has constructs that support modelling of concurrent structures. LOTOS is discussed in a separate report as part of this task.

Jacky reports that writing comprehensive formal specifications is feasible. An unsolved problem is the proper notation for concurency. The author chose Petri nets for concurrency. The specifications for event driven operations were represented by the Software Cost Reduction (SCR) notation [HENI80]. SPECIAL NOTE: The SCR or A7 Notation was developed by David Parnas and colleagues at the Naval Research Laboratory during the late 1970's. During 1989, the SCR technique was used to express the software specifications in support of the safety verification effort for the Darlington Nuclear Power plant in Ontario, Canada [JOAN90]. A recent article by Parnas in the Communications of the ACM cites the use of SCR to safety critical systems [PARN90].

Narayana and Dharap have reported on the successful application of the Z schema to a graphical interface [NARA90A] [NARA90B]. Dialog systems are servers for an interface. They are like operating systems in the concepts they provide. The Z notation was used for the formal design of the system. The authors provide Z schemata for INTERACTOR objects, DIALOG, SCREEN, DISPLAY, PHYSICAL_MOUSE and more.

VII. Object Oriented Design and Reusability

The Z schema provides a concise mathematical notation that can be used to support object oriented design with reusable components. Objects can be decomposed into smaller objects. This supports a top down divide and conquer design procedure. In addition, large objects can be composed of small objects. If the Z schema was used to design the reliable component, the new design will inherit the associated mathematical verification of the component. The Z schema offers an opportunity to reuse reliable software code and its associated verified design specification to compose reliable systems.

This document has been written to propose that the object should be chosen as the viewpoint for a system component. Objects can be decomposed into smaller objects in accordance with a divide and conquer design paradigm. In addition large objects can be composed from smaller reliable objects. Objects can provide a consistent viewpoint across the many phases of specification and design. The Z schema can support the design and reuse of an object. Objects can be used to define reliable and safety critical components and subsystems. Systems can be composed of reliable subsystems. The use of formal methods and the Z schema will provide the assurance of reliability that will be demanded by society for safety critical systems.

VIII. Conclusions and Recommendations

This writer believes that formal methods will be required to specify trusted systems. In the immediate future, software designers should apply discrete mathematics by hand using a proper schema such as Z. This will provide a period of training and application experience for the designers. As the use of formal methods is expanded, it is assumed that tool support will be provided for common schemas. Z editors and proof tools will become available. The reuse of software will be enhanced by the use of formal methods to specify reliable components within an object oriented design paradigm.

IX. Reference Material and Educational Support

Discrete mathematics is the foundation for all formal methods. The mathematics takes the form of propositional and predicate calculus, set theory, relations. functions, and sequences. This writer has found that the 1989 book by Woodcock and Loomes is the easiest to read [WOOD89]. The authors have chosen the Z schema as the syntax to exhibit the formal specification of a telephone exchange.

Ince's book was published in the United States in 1988 [INCE88]. Ince claims to take a gentle approach to the subject of formal specification. The Z schema is chosen to represent the formal concepts. A Z schema formal specification is shown for a library.

The 1981 book by Gries is still valid [GRIE81]. It is one of the required textbooks listed for the Software Engineering Institute course on Software Validation and Verification. This course is now being taught by the University of Houston-Clear Lake in the new and approved Masters of Science degree in Software Engineering Sciences. The technical staff at Odyssey Research Associates in Itacha, New York uses Gries' book as a foundation for their formal mathematics in support of security models and the Penelope Ada verification software engine [GUAS90]. Odyssey has performed this development for the U. S. Air Force Rome Air Development Center.

The book by Spivey presents the syntax of the Z Schema [SPIV89]. One can view Spivey's book as being similar to the Ada Language Reference Manual. It provides the concepts of Z for tool builders. It is not a design book. The book is necessary but it is not sufficient.

Sommerville provides a terse but readable introduction to the concepts of the Z schema [SOMM89]. Sommerville is not teaching discrete mathematics or Z in his book. In the first printing of his book, there are numerical discrepancies between the text and

the Z templates. A sophisticated reader should be able to ignore these minor discrepancies. Several Z schema templates are included in this document.

Reliable systems need to be composed of reliable components. This concept requires that a reuse paradigm be established to incorporate known reliable components with all associated design knowledge and assurances. The Fifth International Workshop on Software Specification and Design was held in Pittsburgh during May 1989. Several papers at this workshop relate to the pragmatic use of formal methods.

The paper by London presents the use of the Z schema to specify two reusable components and their interfaces [LOND89]. Smalltalk was the language used by the investigators.

Another paper proposed that software development may be able to use an analogy to the chemical engineering design process [D'IP89]. D'Ippolito writes "Chemical engineers are not taught to design polypropylene plants; they are taught the unit operations and the objects that provide them ..." . The reader can interpret this idea as composable design using objects and their defined interfaces.

Mary Shaw of the Software Engineering Institute presented another paper that proposes that higher level abstractions are necessary to compose systems from subsystems [SHAW89]. Once again there seems to be a theme that reliable systems are composed of reliable components.

The IEEE used formal methods as a theme for their September 1990 issues of Computer, Software, and Transactions on Software Engineering. The Z schema is well represented in all three publications. Additional articles that have not been cited elsewhere in this reprot are [GERH90A], [HALL90], and [WING90].

X. Local Symposia and Tutorials

The University of Houston-Clear Lake hosted a Software Engineering Symposium on November 7 & 8, 1990. The following tutorials were presented:

Object Oriented Requirement Analysis
Ed Berard, Berard Software Engineering Inc. [BERD90]

Software Reuse
Will Tracz, IBM System Integration Division [TRAC90]

Software Safety
Nancy Leveson, University of California, Irvine [LEVE90]

Applying Formal Methods by Hand
John McHugh, Computational Logic Inc. [McHU90]

Speakers included presentations by the Software Engineering Institute (SEI), the Software Productivity Consortium (SPC), and the Microelectronics and Computer Technology Corporation (MCC). Susan gerhart of MCC provided a review of formal methods with an emphasis on the standards effort that is occurring in the United Kingdom [GERH90B]. The tutorials and presentations exhibited a variety of approaches to produce reliable software. The theme seems to be objects and formal mathematical support for specifying objects.

During 90-91, UHCL and CSC are supporting a seminar series in Information Security, Integrity, and Safety (ISIS). During the late spring or early fall it is planned to invite Odyssey Research Associates to present a tutorial on Formal Methods. Two design examples are used in the presentation:

1) a flight control system
2) secure Network Interface Unit component.

During the summer of 1991, this writer will be teaching a graduate software engineering course on Formal Methods and Models. This course will provide a review of formal methods with an emphasis on a pragmatic application of the Z schema within a case study.

References

[BERD90]
Berard, Ed. "Object Oriented Requirements Analysis." Tutorial, NASA/JSC UHCL 1990 RICIS Symposium: Software Engineering. 7 November 1990. Houston, Texas.

[Deli90]
Delisle, Norman and Garlan, David. "A Formal Specification for an Oscilloscope." IEEE Software. September 1990. pp. 29-36.

[D'IP89]
D"Ippolito, Richard S. and Charles P. Plinta. "Software Development Using Models." Proceedings of the Fifth International Workshop on Software Specification and Design. May 19-20, 1989, Pittsburgh, Pennsylvania. pp. 140-142.

[GERH90A]
Gerhart, Susan L. "Applications of Formal Methods: Developing Virtuoso Software." IEEE Software. September 1990. pp. 7-10.

[GERH90B]
Gerhart, Susan L. "Assessment of Formal Methods for Trustworthy Computer Systems." Proceedings of the NASA/JSC UHCL 1990 RICIS Symposium: Software Engineering. 7&8 November 1990. Houston, Texas.
Dr. Gerhart provided a review of MCC activities and comments on the UK standards efforts.

[GRIE81]
Gries, David. The Science of Programming. New York: Springer-Verlag. 1981. The fifth printing in 1989 is the edition to use.
All known printing errors have been removed. Similar material exists in the more modern book by Woodcock and Loomes cited below. The Gries book can be used to support the Woodcock book in that the UHCL instructors have the set of answers to all questions.

[GUAS90]
Guaspari, David, Carla Marceau, and Wolfgang Polak. "Formal Verification of Ada Programs." IEEE Transactions on Software Engineering. V16 No. 9 September 1990. pp. 1058-1075.

[HALL90]
Hall, Anthony. "Seven Myths of Formal Methods." IEEE Software. September 1990. pp. 11-19.
The author concludes that formal methods must be better understood by developers at large and provides seven facts to replace the seven myths. SPECIAL NOTE: This article was picked as the best of 1990.

[HENI80]
Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application. IEEE Transactions on Software Engineering. SE-6(1):2-13, 1980 (January).
This is the baseline document in the published literature that presents the SCR (A7) methodology as it was applied to the redesign of the Navy A7 aircraft weapons system.

[INCE88]
Ince, D. C. An Introduction to Discrete Mathematics and Formal System Specification. Oxford: Oxford University Press. 1988.
Formal specification example is a library.

[JACK90]
Jonathan Jacky. "Formal Specifications for a Clinical Cyclotron Control System." Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development. 9-11 May 1990, Napa California. pp. 45-54.

[JOAN90]
Joannou, P. "Safety Critical software Satandard Development." Presentation at the Software Productivity Consortium (SPC) 1990.

[LEVE90]
Leveson, Nancy. "Software Safety." Tutorial, NASA/JSC UHCL 1990 RICIS Symposium: Software Engineering. 7 November 1990. Houston, Texas.

[LOND89]
London, R. L. and K. R. Milsted. "Specifying Reusable Components Using Z: Realistic Sets and Dictionaries." Proceedings of the Fifth International Workshop on Software Specification and Design. May 19-20, 1989, Pittsburgh, Pennsylvania. pp. 120-127.

[McHU89]
McHugh, John. "Trusted Systems, Software Engineering and the Internet Worm." Presentation at UHCL. July 14 1989.

[McHU90]
McHugh, John. "Applying Formal Methods by Hand." Tutorial, NASA/JSC UHCL 1990 RICIS Symposium: Software Engineering. 7 November 1990. Houston, Texas.

[NARA90A]
Narayana, K. T. and Sanjeev Dharap. "Formal Specification of a Look Manager." IEEE Transactions on Software Engineering. V16 No. 9 September 1990. pp. 1089-1103.

[NARA90B]
Narayana, K. T. and Sanjeev Dharap. "Invariant Properties in a Dialog System." Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development. 9-11 May 1990, Napa California. pp. 67-79.

[PARN90]
Parnas, David L., John van Schouwen, and Shu Po Kwan. "Evaluation of Safety-Critical Software." Communications of the ACM. June 1990. pp. 636-648.
The most recent technical journal article by Parnas' students. There is still reference to the late 1970's Software Cost Reduction (SCR) work at the Naval Research Laboratory.

[SHAW89]
Shaw, Mary. "Larger Scale Systems Require Higher-Level Abstractions." Proceedings of the Fifth International Workshop on Software Specification and Design. May 19-20, 1989, Pittsburgh, Pennsylvania. pp. 143-147.

[SOMM89]
Sommerville, Ian. Software Engineering, Third Edition. Wokingham: Addison-Wesley. 1989.
A terse but readable introduction to Z is contained in chapter 9.

[SPIV89]
Spivey, J. Michael. The Z Notation. New York: Prentice Hall. 1989.
Think of this book as a reference manual for the syntax similar to the Ada Language Reference Manual.

[SPIV90]
Spivey, J. Michael. "Specifying a Real-Time Kernel." IEEE Software. September 1990. pp. 21-28.
The author provides a report on a case study using the Z notation to specify the kernel for a diagnostic X-ray machine.

[TRAC90]
Tracz, Will. "Software Reuse." Tutorial, NASA/JSC UHCL 1990 RICIS Symposium: Software Engineering. 7 November 1990. Houston, Texas.

[WING90]
Wing, Jeannette. "A Specifier's Introduction to Formal Methods." Computer September 1990. pp. 8-24.
A very readable summary of formal methods with an extensive reference list.

[WOOD89]
Woodcock, Jim and Martin Loomes. Software Engineering Mathematics. Reading: Addison-Wesley. 1989.
The most readable book. The answers to the problems in Gries book act as a support mechanism for this book. A formal specification of a simple telephone exchange is given as the example.

[WOOD90]
Wood, William G. "Application of Formal Methods to system and Software Specification." Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development. 9-11 May 1990, Napa California. pp. 144-146.
This is short presentation on the Software Engineering Institute (SEI) efforts to apply formal methods to the specification of complex systems. SEI has chosen to investigate the path from Z to Annotated Ada (ANNA) to Ada.

Appendix A Comments on Notation

To some software designers the Z notation may seem complex. This short essay is an attempt to put the notation in proper perspective. All engineers have taken calculus and have learned the notation of calculus. The calculus notation is then used in courses such as differential equations, dynamics, strength of materials, thermodynamics, etc. In addition all engineers have learned elements of matrix theory. The result is that most engineers would have not have any difficulty in understanding the notation of the state transition equation:

$$x' = Ax + Bu$$

where ' has been used to represent the derivative with respect to time. Note that Word Perfect has placed a restriction on notation. A overline "dot" is the usual notation for the derivative with respect to time. The other symbols represent:

x - state vector (n by 1)
A - The plant: n by n matrix
u - control vector (m by 1)
B - control laws: (n by m matrix)

The point is that an engineer routinely uses a high level notation to abstract from the fundamentals of the "delta" and scaler notation of calculus. Matrices are represented by single letters without explicit reference to their size. The size is implicit within the context of the equation.

The notation for the Z language is based on discrete mathematics. The attached notation has been extracted from an article contained in the September issue of the IEEE Transactions on Software Engineering [Nara90]. The point is that if an individual claims to be a software engineer, then that individual needs to be familiar with the mathematical notation of software engineering. The Z schema provides a specification template using the notation of discrete mathematics. The notation is not unique to Z.

[Nara90] Narayana, K.T. and Dharap, S. "Formal Specification of a Look Manager." IEEE Transactions on Software Engineering. vol 16 no. 9. September 1990. pp 1089-1103.

Z Notation—Briefly

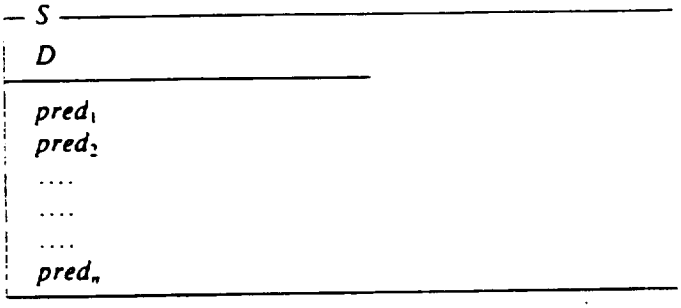
$\triangleq$	"Is defined to be or same as."
$\wedge$	Logical conjunction.
$\vee$	Logical disjunction.
$\Rightarrow$	Logical implication.
$\Leftrightarrow$	Logical equivalence.
$\forall$	Universal quantification.
$\exists$	Existential quantification.
$\mathbb{N}$	The set of natural numbers.
$\mathbb{Z}$	The set of integers.
$m \,..\, n$	The set of natural numbers between m and n inclusive.
$\subseteq$	Set inclusion.
$\subset$	Strict set inclusion.
$\cup$	Set union.
$\cap$	Set intersection.
$\setminus$	Set difference.
$\in$	Set membership.
$\{\}$, $\emptyset$	The empty set.
$\{term \mid pred\}$	The set of terms such that pred.
(a, b)	Ordered pair.
#	The cardinality of a set.
$\mathbb{P}$	Powerset.
disjoint	Pairwise disjoint.
$A \rightarrow B$	The set of total functions from A to B.
$A \nrightarrow B$	The set of partial functions from A to B.
$A \leftrightarrow B$	The set of relations from A to B.
$[a \mapsto b]$	The singleton function ("maplet") which maps a to b.
λ	Lambda abstraction.
$f(x)$	Function f applied to x.
$f\,x$	Function f applied to x.

dom	Domain of.
ran	Range of.
$\fatsemi$	Relational (or functional) composition.
$\circ$	Forward relational or functional composition—$f; g \triangleq g \circ f$.
$R^{\sim}$	Inverse of relation R.
$id\ X$	Identity function on the set X.
R^k	Relation R composed with itself k times.
R^*	Reflexive transitive closure of R.
R^+	Nonreflexive transitve closure of R.
$(\ )$	Image—for $R : A \leftrightarrow B$ and $\overline{A} : \mathbb{P}(A)$, $R(\overline{A}) \triangleq \{b : B \mid (\exists a : \overline{A} \bullet a\ R\ b)\}$.
$S \lhd R$	Restriction of domain of R to S.
$S \lhd\!\!\!- R$	Domain subtraction $S \lhd\!\!\!- R \triangleq X \setminus S \lhd R$ where $S : \mathbb{P}(X)$.
$R \rhd T$	Range restriction of R to T.
$R \rhd\!\!\!- T$	Range subtraction of T.
$R_1 \oplus R_2$	Relation overriding.
seq A	Set of sequences drawn from A.
$\#s$	Length of sequence s.
$\langle\rangle$	Empty sequence.
$s_1 \frown s_2$	Concatenation of s_1 and s_2.
last s	The last element of the sequence s.
front s	All but the last element of the sequence s.
first s	First element of the sequence s.
rest s	All but the first element of the sequence s.

Schema Notations

$S \triangleq [D \mid preds]$ is a schema definition. D is a set of declarations. *preds* are a set of first order formulas on the variables of D. The set constitutes a conjunctive first order formula.

In the graphical notation, a schema has a name S with declarations D and predicate set *pred* and is written



Schema name and the predicate part are optional. Schema names usually designate operations. Components of a schema can be projected using the component names. *pred S* defines the predicate part of the schema S.

1) Combinators on Schemas: A schema can be included in another. The resulting schema contains the union of the declarations and conjunction of the predicate parts.

$S \mid P$	Schema S with P conjoined to the predicate part.
$S; D$	Schema S with the declarations D merged with those of S.
$S[new/old]$	Renaming components: the name *old* in schema S is renamed to *new* every where it appears.
S'	Schema S with all names quoted.
$\neg S$	Schema S with the predicate part negated.
$S \wedge T$	Schema formed by the merging of declarations and the conjoining predicate parts of S and T.
$S \vee T$	Schema formed by the merging of declarations and the disjoining of predicate parts of S and T.
pre S	*Precondition:* All the state after the dashed variables and outputs are hidden.
post S	*Postcondition:* All the state components undashed variables and the inputs are hidden from the predicate part.
$S \oplus T$	Schema overriding $\triangleq S \wedge \neg pre\ T \vee T$.
$S; T$	*Schema Composition:* Schema formed when the final states of S become the initial states of T: in such composition, the base names of the dashed identifiers in S and the matching undashed identifiers in T are renamed and existentially quantified in the predicate part.

2) Conventions: The following conventions are used for variables and predicates in schemas which designate operations.

undashed	State before the operation.
dashed	State after the operation.
ending in ?	Inputs to the operation.
ending in !	Outputs of operation.

$S \Rightarrow T$	Schema formed by merging declarations and taking *pred* $S \Rightarrow$ *pred* T as the predicate part.
$S \setminus (v_1, \cdots, v_n)$	*Hiding:* Schema formed by removing declarations $v_1, \cdots, v_n$ in S and existentially quantifying the predicate part of S with respect to those variables.

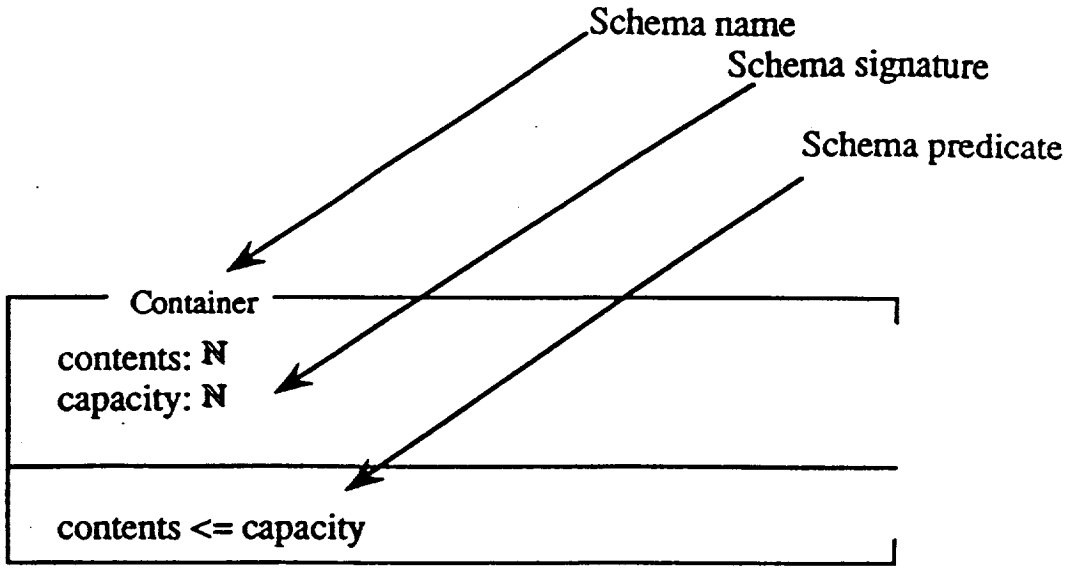
Appendix B Z Schema Templates

These templates have been reproduced from [SOMM89]. They are shown to demonstrate how low level schemata can be incorporated into higher level schemata within a terse but readable format.

The low level Container and Indicator schemata (objects) have been used in the definition of the Hopper schema. The expanded version of the Hopper schema is shown to exhibit the total definition that results from including the Container and Indicator signatures into the Hopper definition.

The SafeFillHopper is a schema for a modified fill operation. The delta on the Hopper means that this operation will change the state of the Hopper. The OverFillHopper schema is another example of an operation. The FillHopperOP has a predicate that provides an error check on the fill operation.

[SOMM89]
Sommerville, Ian. Software Engineering, Third Edition. Wokingham: Addison-Wesley, 1989.



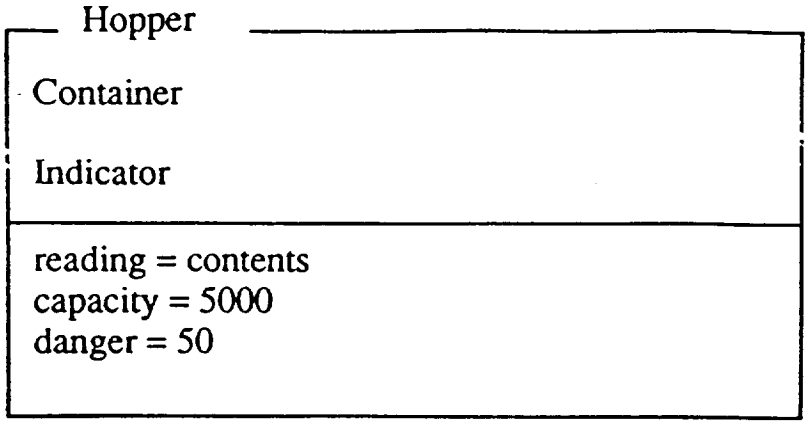
A Z(Zed) schema

Indicator

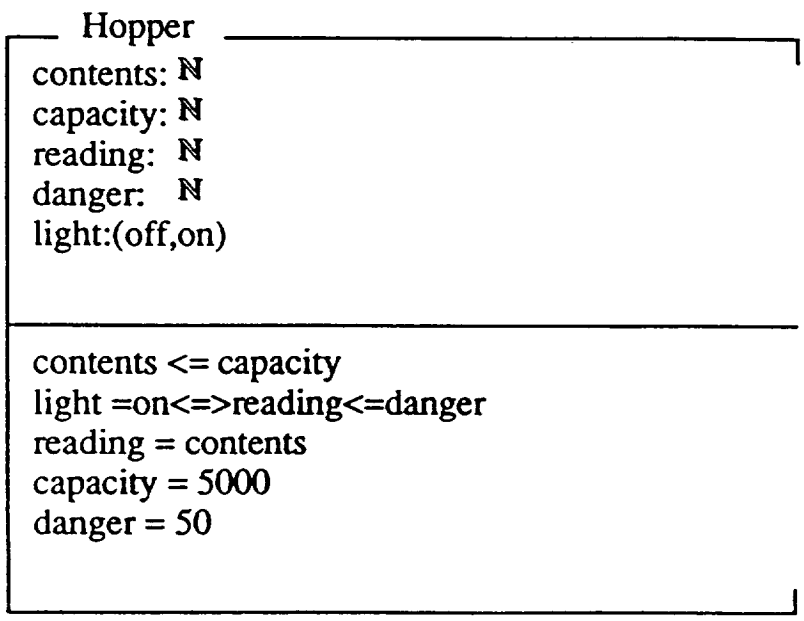
light:{off,on}
reading: ℕ
danger: ℕ

light = on <=> reading <= danger

The specification of an indicator.



The specification of a hopper.



The expanded specification of a hopper.

SafeFillHopper

Δ Hopper

amount?: $\mathbb{N}$

---

contents + amount? <= capacity

contents' = contents + amount?

The specification of hopper fill operation avoiding overflow.

OverfillHopper

Δ Hopper
amount?: $\mathbb{N}$
r! = seq CHAR

---

capacity < contents + amount?
contents = contents'
r! = "Hopper overflow"

The specification of the hopper overflowing.

FillHopperOP

SafeFillHopper v Overfill Hopper

The specification of the hopper filling with error check.