Applying the B Technologies to CICS

Jonathan Hoare[1]
Jeremy Dick, Dave Neilson and Ib Sørensen[2]

[1] IBM UK Limited, Hursley
[2] B-Core (UK) Ltd., Oxford Science Park

Abstract. This paper[3] reports on the experiences of IBM Hursley in using the Z notation and the B-Method [Abr95] [Abr93] in developing new function for IBM's CICS product[4] [IBM94].
A major constraint on the project was the need to produce code that not only corresponded to its required function, but also met a number of stringent non-functional requirements in areas such as integration, performance and maintenance.
The Z notation was used to capture the required function, and the resulting specification was hand-translated into AMN. The B-Toolkit, with project-specific extensions, was then used for the development down to PL/X code.
The success of this endeavour is discussed here. The use of Z and the B-Method were very successful in addressing the new functional requirements. Meeting the non-functional requirements, however, was more difficult.

1 Introduction

The application of mathematical methods to the development of IBM's CICS products began in 1982 with a collaboration between Hursley and the Programming Research Group at Oxford University on the use of the Z notation [Spi89] [Wor92] for the specification. Initially Z was used in the CICS code restructuring initiative [CN88] and since then it has been used for the specification of most new components of CICS as well as parts of the API [CNS89].

In this Z based development, whilst the specification stage was formal, the rest of the process did not change. The aim of IBM's use of the B-Method [NH94] is to reduce the effort in design and coding by extending the use of mathematical techniques and associated tools to cover these later stages, consequently allowing developers to concentrate on the creative aspects of software design.

The current project began in March 1993, when a site licence for the B-Toolkit [B-C93] was obtained for Hursley, and a collaborative contract was set up with B-Core (UK) Ltd..

[3] Parts of this paper are based on [Hoa95].
[4] CICS is a trademark of IBM Corporation.

2 The B-Method and B-Toolkit

The B-Method [NH94] [ALN+91] [Abr95] is a state-based method which uses the Abstract Machine Notation (AMN). Invented by J-R Abrial, the notation is based broadly on the same mathematics as Z (Zermelo set-theory and predicate logic), extended by a generalised notion of substitution. It contains a very small imperative programming language augmented by a number of specification constructs for expressing non-determinism. AMN semantics is compact and thus tractable for sizable developments.

AMN does not address concurrency. The aspects of systems addressed by the B-Method are similar to those of Z and VDM.

AMN admits three constructs: **MACHINES** which are expressed using only specification constructs, **REFINEMENTS** which may mix specification constructs with imperative statements, and **IMPLEMENTATIONS** in which all non-determinism is resolved through the use of imperative constructs only. Implementations are very easy to translate into commonly used programming languages.

The B-Method encourages software development in layers. Top level specifications are implemented in terms of lower level specifications, or designs which contain more detail. Thus the process of refinement decomposes abstract machines into smaller machines which are separately implementable through full information hiding. The verification conditions that must be proven for correct refinements are stipulated by the method.

The B-Toolkit [B-C93] provides a suite of programs that support the B-Method. These include a parser, typechecker, animator, proof obligation generator, proof tools and code translators. The three main products of a B development are code, design documentation and proof listings. The tools co-operate in a rigorously controlled programming environment driven by a dependency manager, which ensures the consistency of these three main products.

A major feature of the project reported in this paper is the extension of the B-Toolkit to accommodate the CICS development environment. These extensions are described in detail in Section 5.

The B-Method and B-Toolkit were selected as they were considered at the time to represent the only practical approach to formal refinement.

3 Approach

This project began with what has become standard practice in the CICS organisation: the production of an initial requirements document called the *Product Level Design* (PLD) that laid out the requirements which were to be addressed by the proposed component and included an abstract Z specification of the new function which was to be provided. Also included were details of some non-functional characteristics that needed agreement prior to the design phase.

In order to link into the B-Method, the Z specification in the PLD was rewritten in AMN, and the B-Toolkit animator was used to validate the hand translation. A certain amount of proof work was also carried out at the specification level at this stage.

An AMN design was then created to prototype the system against a model of the underlying API consisting of a set of C libraries. C code was generated from the prototype design, run and tested.

Satisfied with the overall design, the development team then proceeded to redevelop towards the final target language, PL/X, a PL/1-like IBM proprietary systems programming language. To achieve this, a new set of B-Toolkit system libraries had to be hand-coded in PL/X to form the target of the development.

Using a specially developed translator, PL/X was generated as the final code. This was subject to unit test, integration test and system test.

Figure 1 summarises the approach taken, showing the dependencies in the various activities undertaken.

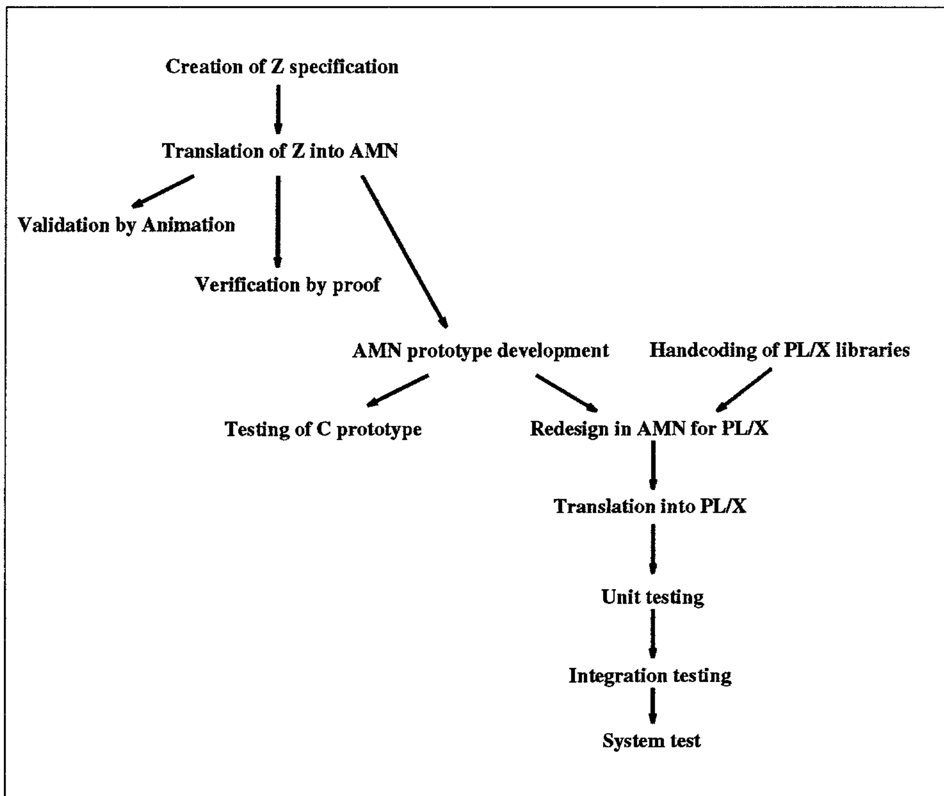

Fig. 1. The development approach adopted.

4 Non-functional Requirements

With an installed customer base of in excess of 30,000 licences, CICS is a mature product with a well established architecture and support infrastructure to which any further development must conform. For the application of a new method to CICS development, as described in this paper, this presents a number of challenges.

Several important non-functional requirements are not normally stated in the PLD, but are implicit for any new development. Amongst these are the following:

Architecture. The CICS architecture imposes a number of constraints, including the use of PL/X as the target language, CDURUN[5] definitions for interfaces between components, CICS storage management facilities (as opposed to native operating system functions), and the general use of CICS facilities for synchronisation and concurrency control.

Size. Various limits on source and object code size for CICS modules are imposed by tools used in CICS development, such as the library system and compilers. CICS standards and architecture also set bounds on certain parameters, such as automatic storage. In addition, code size can have implications for execution performance of the resulting programs.

Performance. Typically, a CICS component will have many operations, a small number of which are identifiable as performance critical. It is important that the modules implementing these operations are tuned to achieve a high degree of efficiency.

Maintenance. An extensive maintenance process for CICS (involving hundreds of service personnel, field engineers and others) is in place, and it would be unrealistic to expect a radically new approach to be adopted. Code generated in any development must conform to existing maintenance standards, including the use of standard trace and debugging facilities.

Enhancement. Any new development of CICS must form a sound basis for future extensions in later releases, including the ability to add new function to an existing area or modify the design if necessary. It is therefore essential that the designs, and code where appropriate, are in a form which would allow such extensions to be made readily.

5 Addressing Non-functional Requirements

An essential feature of this project has been the availability of a comprehensive suite of tools to support each stage of the development process. A number of tools specific to the development environment were created in parallel with the CICS design work and there has been a significant amount of feedback from the project in this area, influencing not only the evolution of the CICS specific tools, but also parts of the base toolkit.

This section describes those elements of the toolkit which were developed in order to meet specific non-functional requirements.

[5] A high-level language for the definition of CICS internal interfaces.

The PL/X translators and library coding tools.

An implicit non-functional requirement was that the IBM internal PL/X language should be used for the eventual code, and so an appropriate translator for AMN implementations was incorporated into the toolkit. This tool incorporated program transformation to improve the quality of the final code. A coding tool was also provided to assist in the production of the lowest level library modules.

Control Block generator.

The base B-Toolkit includes a utility called the base generator to produce a complex specification and its AMN implementation from a high level description of the encapsulated state, written in a special definition language. A similar facility was required for CICS whereby the software developer can give an abstract description of the control blocks needed in his design. The control block generator takes this description, and produces a specification which encapsulates access to the control block, together with a corresponding PL/X implementation. This was required in order to use the standard CICS facilities for storage management.

CDURUN generator.

The internal interfaces between CICS components (domains) are defined using a special language called CDURUN. In order for other parts of CICS to be able to use a new component it is necessary to produce a CDURUN definition for the top level interface. To meet this requirement a generator tool was written which produces a CDURUN definition and an AMN specification from a high level description of the interface. An output from this tool is also used to determine the action of the final CICS module construction tool.

Module construction.

The physical organisation of code into modules in the CICS environment often does not correspond to the layered logical structure of a B development. Instead, a vertical partitioning is used, in which all the PL/X code needed to implement a group of top-level operations is collected into a single compilable file or module. The CICS module construction tool performs this vertical partitioning for a B development by restructuring the B development tree. Such a vertical partitioning improves the performance of the final code by localising the flow of control.

Trace formatting.

For debugging purposes, particularly during maintenance, CICS standards require execution trace entries to be appended to a log file. An offline utility program is used to interpret the trace entries, allowing CICS' internal processing to be reconstructed. A facility was added to the B-Toolkit to add trace entries into the PL/X generated from AMN implementations, and to produce the corresponding trace interpretation code.

Dump formatting.

If an error occurs during CICS execution, or if explicitly requested, a storage dump is written to disk. For maintenance purposes, an offline utility program

is used to reconstruct the state of each CICS component at the time that the dump was taken. To meet this requirement, a facility was added to the B-Toolkit to generate code automatically for this utility.

Code annotation.
CICS maintenance requires code to be well annotated. The B-Toolkit translation process was extended by adding facilities that produce automatic annotation, hence increasing the traceability of program fragments back to AMN designs.

The GML markup tool.
The base B-Toolkit produces formatted output documents in the form of LaTeX source. IBM, however, uses its own proprietary markup language, GML [IBM89], for its documentation. It was therefore necessary to add an option to the toolkit to produce GML output instead of LaTeX.

The CICS specific tools were incorporated into the same dependency and configuration management environment as that of the base tools in the B-Toolkit.

Figure 2 shows the interaction of the tools specific to this project and the entities that they exchange. Processes appear as shaded diamond shapes; objects acted upon by those processes as rectangles. The rounded box marked 'Layered Design Process' embodies all the other work carried out within the base tools of the B-Toolkit.

The ease with which new tools could be added to the B-Toolkit was remarkable. This extensibility is achieved in such a way that all tools constitute a complete integrated development environment. The ability to so extend the B-Method and supporting tools is a direct consequence of working with a uniform syntax in all kernel tools.

6 Assessment of Results

The estimate at the outset of the project for the number of PL/X statements required to implement the specified function was 15 KPS[6], whereas the B development resulted in 20.4 KPS of PL/X. This last figure includes code that is replicated as a result of the vertical partitioning used during module construction, estimated to be approximately 35% ot the total. Normalising by this factor gives a figure of 13.3 KPS. Effort expended on the development amounted to 30 man-months (not including tool development).

The B development was structured into layers comprising 61 AMN constructs (abstract machines and implementations). The top layer specification contained 30 operations, fanning out into 324 distinct operations in the overall development. Four control blocks were used to define the concrete data structures, containing a total of 70 data fields. The generated PL/X comprised 8 modules.

Despite the complexity of the project, the B-Method proved highly satisfactory in meeting the functional requirements. Complete functionality was achieved down to running code, first in C in a simulated environment, and then

[6] 1000 program statements.

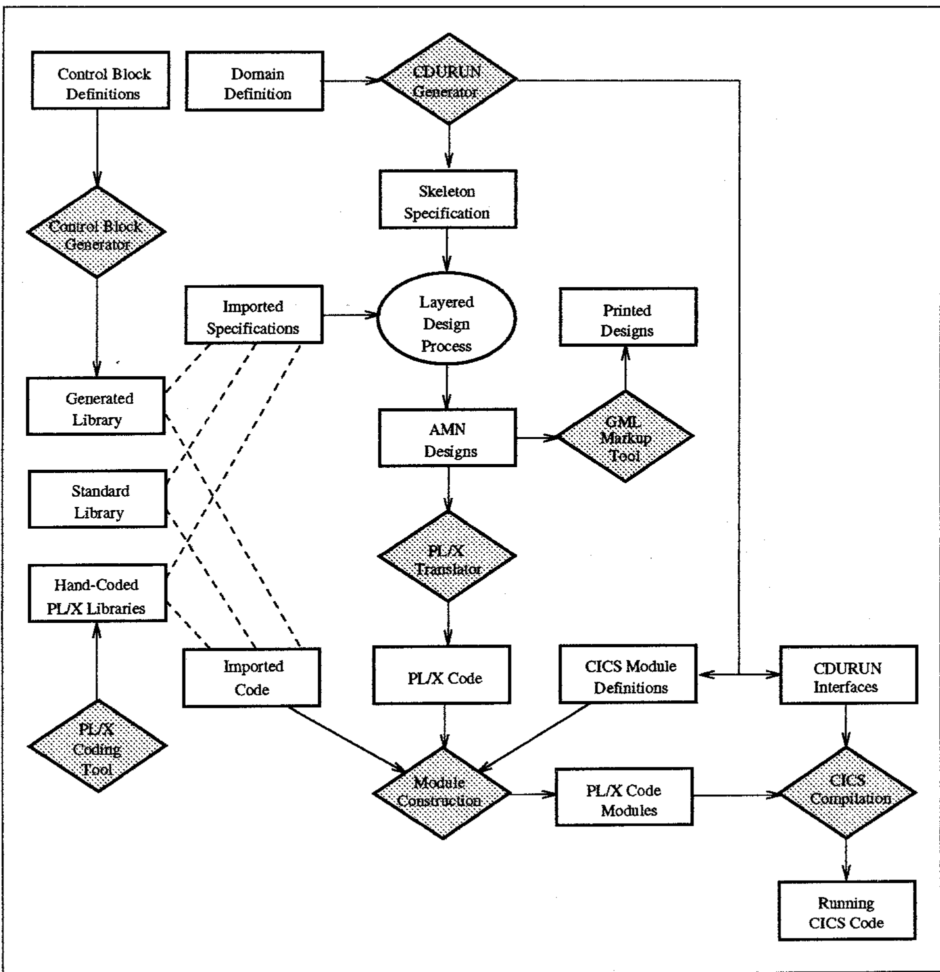

Fig. 2. Organisation of the CICS-specific tools.

in PL/X in the CICS environment. No data design errors were found in the AMN, no coding errors were produced in the code generated from AMN, and only easily correctable errors of omission in algorithmic design were encountered.

A summary of the numbers of faults found in each development activity is found in the Fault Grid [Dic95] in Figure 3.

The clusters of errors on the diagram reveal the activities that are prone to introducing errors. These are as follows.

- The creation of the original Z specification, which was carried out with no tool support apart from a type-checker. The bulk of these errors where found

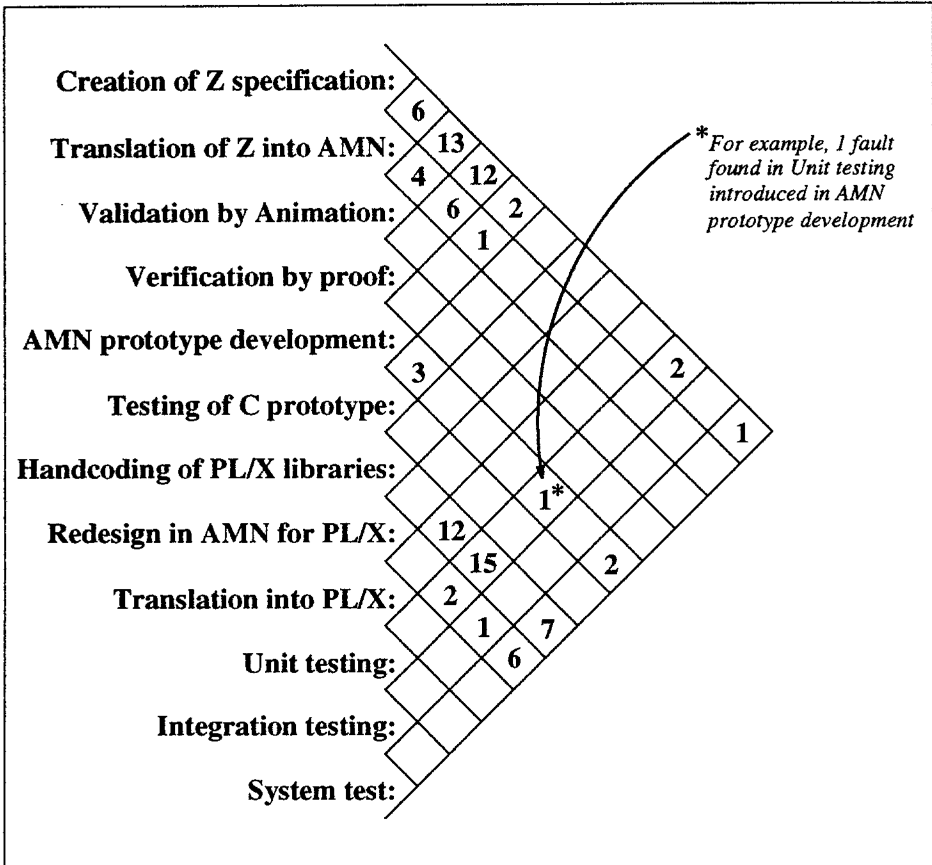

Fig. 3. Fault Grid for the CICS deveopment.

by animation and proof work performed on the AMN after translation. This underlines the importance of such tools in assessing the quality of a specification.

- The hand-coding of the PL/X libraries, which was carried out with limited tool support (in the form of the PL/X Coding Tool).

Note in particular the small number of faults found during testing of the that related to the original specification, three in total. The one fault at the peak of the grid (where the correction of errors is the most costly) was concerned with the invocation of an operation outside of its precondition.

The integration requirements of the CICS architecture were met successfully by the creation of three specific tools: the Control Block Generator, the CDU-

RUN Generator and the Module Construction tool.

The ability to produce code semi-automatically for dump and trace analysis tools was considered to be a distinct asset, and saved considerable effort in comparison with hand-coding. Also, the propagation of design annotations into the code, including traceability information, reduced programming effort.

In the final review, however, the code fell short of satisfying the non-functional requirements detailed below.

Size. The generated code exceeded acceptable limits for the CICS environment in several respects:

- The call-by-value parameter passing mechanism increases the requirement for automatic storage, and consequently causes the CICS architecture limits to be exceeded in some situations. More insight into how to use call-by-reference mechanisms is required before program transformation techniques can be used successfully to address this problem.
- Greater code size implies an increase in the number of variables, and a corresponding reduction in the ability of the PL/X compiler to make sensible use of base registers for optimisation purposes. This problem has been partly addressed by adding program transformations which remove many local variables used to pass values between operations.
- The volume of automatically generated annotation in the PL/X code made the source files too large for the CICS library system to handle. This problem could be addressed by finding the right balance in the amount of annotation generated.

Performance. The final code was not considered to be adequately efficient for certain critical operations. The reasons for this were:

- The hierarchical design adopted in the project is reflected in the structure of the final code. Where an operation has to access two distinct encapsulated data items through sub-operations, algorithmic control cannot descend below the lowest common ancestor of the two items. The effect of this is that control has to repeatedly descend and ascend through a series of layers to effect the operation, thus introducing inefficiency. It is possible that an alternative design could have been found to alleviate this problem. The addition of multiple refinement to the B-Method could also help in this area.
- Call-by-value is used in the generated code, sometimes making it necessary to copy quite large structures on calling functions, with consequent inefficiency. Program transformations applied at the code generation stage went some way to solving this problem by removing redundant parameter passing.
- Manual optimisation techniques, which would typically be used in association with traditional development, were not applied. Some of these techniques can be automated, but others, by their very nature, are not automatable, since they require an empirical approach. Manual post-hoc modification of generated code is undesirable in that it invalidates the AMN design, hence undermining many of the benefits of the method.

Maintenance / Enhancement. The code was not considered acceptable for maintenance and extensibility according to CICS standards, for the following reasons:

- The code, even with annotations, was not comprehensible enough to make it modifiable with confidence. Here, efficiency considerations in the use of in-line code to avoid function calls adversely affected readability.
- Modifications to the PL/X would have to be factored back into the AMN. For external logistical reasons, the decision was taken only to maintain the code at the PL/X source level.
- Logistical concerns raised issues regarding the longevity of the tools and the maintenance of relevant skills.

7 Recommendations

Given the stringent constraints imposed by the development environment, a compromise between traditional methods and the full B-Method would have been more suitable.

Our recommendation would be to use the base generators and libraries to produce an abstract executable prototype in which all major design decisions are cleanly expressed. Without the need to include implementation details of no relevance to the fundamental data or algorithmic design, there would be more freedom to remain abstract.

The prototype should be developed using the full B-Method from specification to low-level design, using typechecking and animation at every stage as a means of validation. Verification by proof should be carried out on selected modules.

When satisfied with the design, the development should be completed by traditional coding methods in compliance with the external requirements, working from the lowest level formal design of the prototype.

8 Conclusion

Never before have tool-supported formal methods been applied to this extent on an IBM project of such complexity. A major contributory factor in overcoming the considerable challenges encountered was the commitment of the project team working in close cooperation with the tool suppliers.

Whilst the meeting of non-functional constraints was a considerable challenge, overall the application of the B-Method in the CICS project was highly successful in addressing the functional requirements of the development.

Software engineering methods and tools often ignore the problem of non-functional requirements. Indeed, it is difficult to conceive of a generic tool sufficiently flexible in its approach to cater for the diversity of external constraints imposed by existing development environments.

For new developments, however, particularly those without such predetermined constraints, this project has demonstrated the feasibility of a tool-supported

rigorous development, encompassing formal specification through to automatic translation into code.

References

[Abr93] J.R. Abrial. B-Technology technical overview. 1993.

[Abr95] J.R. Abrial. *The B-Book - Assigning Programs to Meanings.* 1995.

[ALN+91] J.R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, and I.H. Sorensen. The B-method (software development). In W.J. Prehn, S.; Toetenel, editor, *VDM 91. Formal Software Development Methods. 4th International Symposium of VDM Europe Proceedings.*, volume 2, pages 398–405. BP Res., Sunbury Res. Centre, Sunbury-on-Thames, UK, Springer-Verlag, Berlin, Germany, October 1991.

The B-method is a formal software development process for the production of highly reliable, portable and maintainable software which is verifiably correct with respect to its functional specification. The method uses the abstract machine notation (AMN) as the language for specification, design and implementation within the process. AMN is a sugared and extended version of E.W. Dijkstra's (1976), guarded command notation with built-in structuring mechanisms for the construction of large systems. The method is supported over the entire spectrum of activities from specification to implementation by a set of computer-aided tools. (3 Refs).

[B-C93] B-Core (UK) Ltd. B-Toolkit User Manual. *(available from B-Core (UK) Ltd. on request)*, 1993.

[CN88] B. P. Collins and C. J. Nix. The use of software engineering, including the Z notation in the development of CICS. *Quality Assurance*, 14(3), September 1988.

[CNS89] B. P. Collins, J. E. Nicholls, and I. H. Sorensen. Introducing formal methods: The CICS experience with Z. *IBM Technical Report TR12.260*, 1989.

[Dic95] Dick, J. Fault grids: another way of presenting fault counts. *Software Reliability and Metrics Club Newsletter*, (16), 1995.

[Hoa95] J. P. Hoare. Application of the B-Method to CICS. In J. P. Bowen and M. Hinchey, editors, *Applications of Formal Methods.* Prentice Hall International, 1995.

[IBM89] IBM Corporation. *The IBM Publishing Systems BookMaster General Information. Manual GC34-5006*, 1989.

[IBM94] IBM Corporation. *CICS Family: General Information. Manual GC33-0155*, 1994.

[NH94] D. S. Neilson and Sorensen I. H. The B-technologies: A system for computer aided programming. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory*, pages 18–35. B-Core (UK) Ltd., BRICS Notes Series, Univ. Aarhus, Denmark, 17-19 October 1994.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Hemel Hempstead, 1989.

[Wor92] J. B. Wordsworth. *Software Development with Z: a Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley, 1992.