Technical Report

UCAM-CL-TR-188
ISSN 1476-2986

Number 188

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Specification of computer architectures: a survey and annotated bibliography

Timothy E. Leonard

January 1990

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

I first define computer architecture and architecture specification, explain how the conflict between clarity and ambiguity makes writing specifications difficult, and introduce and consider the advantages and problems of formal specifications. I then survey all the literature on architecture specification, and introduce the literature on technical writing and on formal specification in general. I close with an annotated bibliography.

Contents

Introduction

Computer Architectures — The term "computer architecture" is used with two distinct meanings. In the broader sense, computer architecture is an aspect of computer systems design: creating an architecture consists of dividing a system into major components and defining the interfaces between those components. An architecture is thus the overall structure of a system. In the narrower sense, computer architecture is the design of the interface that is unique to computers: the program interface. (The program interface is unique in that anything with a program interface is a computer, and anything without a program interface is not a computer.) In this paper, I will use the term in the narrower sense: as the name of the interface between programs and an interpreter of programs.

Interfaces — An architecture, then, is the definition of an interface.[1] It defines a set of objects (each of which may have some state); defines a set of operations for manipulating and observing the objects and their state; says what initial states are allowed and what inputs and operations are allowed from each state; and says what results and resulting states are allowed from each operation. The definition must not contradict itself, and it must be complete. It needn't be a function—it may allow an operation to return any of several results, or even any possible result—but for every possible operation it must say what results are allowed. These conditions are met by any full definition of an interface. The importance of completeness is twofold. First, a fully defined architecture can stand on its own; the architect has made all the necessary decisions so other people can implement or use the interface. Second, a fully defined interface is a formal mathematical system.[2] Because a fully defined architecture is a formal system, many of the tools of mathematics can be used to analyse, manipulate, and express it.

---

[1] Actually, an architecture is often the definition of two interfaces; one for software and one for processors. Typically, the two are identical except for operations reserved for future use, which software is prohibited from depending on, but to which processors are required to react in a specific way (for example, by reporting an exception).

[2] For example, an architecture's units of state can be considered an alphabet, the rules defining the allowed initial states and inputs can be considered axioms, and the rules defining the allowed results and outputs can be considered primitive rules of inference. Similar correspondences can be made to other kinds of formal systems.

Interpretation of Implementations — Although it is common to say that an implementation conforms to an architecture specification (or, more simply, that it meets the architecture), this should actually be qualified by adding, *with respect to a particular interpretation.* An interpretation says which aspects of the implementation are to be taken as corresponding to the objects and operations of the architecture. For example, the VAX-11/750 has a HALT button on the front panel that is clearly intended to be interpreted as invoking the HALT operation described by the VAX architecture. Because the intended interpretation is often so clear, it is almost always ignored as being obvious. But the interpretation must be made explicit for two reasons. First, a system's correctness depends on the correct interpretation of each lower layer in the terms of the layer above. It's quite possible for an interpretation to be wrong, even if the implementation of each layer is correct, and then the system is broken. Second, it is only the need for an explicit definition of an interpretation that generates a need for an explicit definition of the architecture's observables, yet unless they are defined, the architecture is incomplete. In practice, architectures' observables are often not defined, and this causes problems.

Difficulties of Architecture Specification — Once an architecture has been designed, it needs to be specified. Typically, the specification is carefully written in a natural language, sometimes augmented with a complete or partial example of an interpreter written in a programming language. This specification needs to be both easy to understand and completely unambiguous. For the intended audience (computer engineers and programmers) such descriptions are normally easy to understand. Unfortunately, natural-language descriptions are notoriously ambiguous, and the harder one tries to make them unambiguous, the harder to understand they become. Conversely, formal language descriptions (including programming-language descriptions), can be made very unambiguous, but are difficult to understand and can have other problems, perhaps the best-known of which is overspecification. The difficulty, then, is writing a specification that is simultaneously easy to understand and sufficiently unambiguous.

Formal Systems — Because an architecture is a formal system, many mathematical systems can be used to express it. The constraints defined by the architecture can be directly expressed in the form of axioms, or alternatively, the same constraints can be expressed in terms of a second formal system (a meta-theory). Many different mathematical systems are being used as meta-theories for describing architectures, including first-order and higher-order logic, set theory, temporal logic, and many others. The systems differ in expressive power, in expressive style, and in the power of the mathematical reasoning they support. The arguments for low-powered formal systems are that the mathematics are easy to understand; the proof tools are powerful (in general, the weaker the logic, the more a proof tool can accomplish automatically); the logic forces the specification to have desirable properties, like consistency or executability; or that the specification can be automatically implemented. The arguments for high-powered formal systems are that the mathematics are able to express

whatever is needed now or in the future, and that it is possible to express concepts more directly. The proper choice depends on each specification's intended use. Since the primary requirement of an architecture specification must be that it correctly captures the architect's intents, expressive power is very important. The closer the mathematical expressions can come to the informal requirements the architect has in mind, the better. This favors more powerful mathematical systems. If a weaker system is necessary in order to support automatic implementation, for example, a second specification, written in the weaker system, can be written and formally proved to be equivalent to the primary specification (or a restriction of it).

Specification Structure — Each formal system brings with it a style of thinking and a style of structuring specifications. While it is possible to structure architecture specifications in a variety of ways, most often a fruitful way of thinking has already been incorporated into the informal concepts, representations, and methods in use by people working in the field. The safest structure for a formal specification is one that directly corresponds to these informal concepts and representations already in use. (It's for this reason that a clean architecture is easier to specify than a messy one—a clean design has fewer concepts to formalize.) What's more, such a formal specification will be the easiest to understand for people already trained in the field. Informal concepts and representations may, however, be poorly thought out, and considerable effort may be necessary to make them simpler and clearer before they can be formalized. On the other hand, the resulting clarifications may be more useful than the formal specification itself.

Notation — A formal specification must be expressed in some notation. It is possible to express formal concepts in natural language, but there are several difficulties with doing so, including verbosity and the inevitable confusions between the formal and the usual meanings of the same terms. A good notation can draw attention to important statements, relationships, and symmetries, can make it easy to ignore irrelevant detail, and can make it easy to summarize important distinctions compactly. When choosing a formal system it is important to remember to distinguish between the system itself and the notation used to express it. It's much easier to change notations than systems.

Specification Maintenance — Complex specifications are difficult to write, understand, and maintain. The discipline of software engineering includes theories and methods for developing, understanding, and maintaining software systems. Since programs are specifications, many of the theories and methods of software engineering apply just as well to specifications in general, even nonexecutable ones. So, for example, modularity, abstraction, and conceptual integrity are as important to formal specifications as to programs, and many of the techniques and tools that software engineers use will help formal specifiers as well.

Specification Testing — Because an architect's intentions are not directly accessible as a formal system, there is no way of guaranteeing that a formal specification is what its designer intends. That is, a specification can have bugs. In the end, the comparison of a formal system to its designer's intentions is inherently informal, and the most obvious method is to have the architect review specifications for correctness. Formal methods can't replace such reviews, but they can assist. For example, it may be possible to prove that a specification is self-consistent, or that it has other expected properties, or that an example implementation meets the architecture, or that the architecture can be used to build a larger system. In general, formal proofs can increase confidence in a formal specification in the same way that test executions increase confidence in a program. A formal specification must be tested (by test proofs) before it is used, and the results of the tests must be informally reviewed for correctness.[3]

Specifications as Documentation — In order to be easy to understand and easy to use, a formal specification needs to be integrated with supporting informal structures. For example, a good specification should include an introduction; a description of the scope and purpose of the specification; a list of related documents; tables, figures, and illustrations; example implementations and example uses; and reference aids like a table of contents, glossary, and index. Note that though a specification should be supported by this additional material, the formal specification itself should either be expressed in a notation that is distinct from natural language, or clearly distinguished in some way to separate it from the commentary and reference aids. Readers must be able to unambiguously determine whether a statement is specification or commentary, because implementations that contradict the commentary may be allowed, but implementations that contradict the specification are not allowed. If the specification is expressed twice, once in an unambiguous formal notation and once in an easy-to-understand natural language, one of the two must be explicitly declared the actual specification, the other should be derived from it, and great care should be taken to ensure that the two are equivalent and that they stay equivalent as the specification evolves.

Summary In summary then, a computer architecture is the definition of a programmable interface. In developing an architecture, the architect chooses a set of objects and operations, decides what operations are allowed in what states, and decides what is observable and what results are allowed. In recording the architecture, the specifier chooses a formal mathematical system, a specification style, and an expressive notation. The formal system must be sufficiently powerful to directly express the architect's intents. The specification style should correspond to the informal style already in use, and should be structured so as to be easy to understand and easy to maintain. The notation should emphasize the important parts of the specification and minimize the irrelevant parts. The

[3]There is an exactly corresponding problem at the concrete level: ensuring that an implementation's design (which is a formal object) is a valid interpretation of a physical machine (the *realization*). This validity can be checked only informally.

specification can be managed with the techniques of software engineering, and must be tested and debugged. Finally, the formal expression of the specification should be integrated with informal aids to make it easy to understand and use.

Survey

Architecture Specification — [Wright] lays out the problems of architecture specification as documentation and describes IBM's architecture documentation process. He distinguishes between designing an architecture and documenting it, lists and defines several requirements of an architecture specification (it must be understandable, well written, exact, complete, and unified), and explains how IBM's documentation process is designed to achieve each of these requirements.

Iverson argues for using APL as an architecture-specification language, and demonstrates it on IBM's 7090 [Iverson], and IBM's System/360 [Falkoff] architectures. [Case] reviews the history and significance of the System/370 family, and includes a section describing IBM's architecture control procedure. In [Gifford], Case and Padegs describe the architecture's scope, evolution, future, description, maintenance, and control. In particular, they mention that IBM does not use APL for architecture specification because they feel that formal languages are not flexible enough to express all they need to say, because the language can't be understood and used by the intended audience without training, and because of the difficulty of keeping formal and informal specifications consistent with each other.

[Bhandarkar] briefly describes architecture management at Digital, including problems of architecture specification.

Bell and Newell [Bell,Siewiorek] complain about the often poor quality of architecture description, and justify and introduce ISP. ISP builds on prior work at Carnegie-Mellon University on architecture specification [Haney,Darringer]. It has achieved widespread use for a variety of purposes [Barbacci81], and is the only architecture description language to have done so. [Barbacci79] discusses architecture description and describes experience at it using ISP. [Parker] describes problems that users have discovered with ISP, and lists requirements for an ideal architecture-description language, including the need for a formal semantics.

[Bowen87] argues for the use of a formal language for architecture specification, and demonstrates the use of Z. The group at Oxford also specifies the Motorola 6800 architecture [Bowen86], parts of the Motorola 68000 architecture [Rose], the inmos transputer architecture [Bowen89,Farr], and IEEE floating point [Barrett].

[Geser] specifies the intel 8085 architecture as an abstract data type using an algebraic specification language.

There are many examples of architecture specifications written in programming languages or computer-hardwaredescription languages, as in [Chen], [Eichenseher], and those referred to in [Dasgupta82]. In almost all cases, the architecture's observables are not defined, and in most cases the specification language is not given a formal semantics. [Dasgupta84,Dasgupta85] and [Damm] are notable counterexamples in that their languages are proposed as architecture-specification languages and are given Floyd-Hoare style axiomatic semantics.

There are literally thousands of architectures informally described in the literature. Of them, [IBM] is probably the most carefully written. For a broader range, [Siewiorek] provides descriptions of about 40 machines, and provides references to a great many more. Although the descriptions are not particularly recent, the sample is still representative since almost all more recent descriptions use the same old styles. Two interesting recent specifications are [inmos], which describes a machine that is formally specified by [Bowen89,Farr], and [Kershaw], which describes a machine that is formally specified in [Cullyer] and is formally specified and partially verified in [Cohn87,Cohn88].

Writing Well — In order for an architecture description to be clear, it must be written well. There is an immense literature about writing well. [Mosenthal] introduces recent research. [Strunk] is a classic short handbook of style. [Chicago] and [Skillin] are classic comprehensive style guides. [Browning], [Kuehne], and [Stephen] cover the issues and techniques of computer documentation. [Carlson] provides an annotated bibliography on technical writing, and [TechComm] is a journal devoted to technical writing.

Formal Specification — [Melliar79] provides an excellent introduction to formal specification and surveys the literature from before 1979, and [Cohen86b] provides more recent coverage. See also [Harman] and [Parnas]. For arguments in support of formal specifications, see [Dijkstra], [Guttag80], [Horning], [Liskov], and [Meyer]. [Guttag80] describes the use of trial proofs in testing a formal specification, and argues that the clarifications produced by formal specification are more useful than the specification itself. [Berztiss] covers the history and literature of abstract type specification. Much of the work on formal specification has been directed toward formal verification or synthesis. For descriptions of formal verifications of entire systems, see [Melliar82] and [Bevier87]. For a discussion of the meaning and limits of verification, see [Cohn89]. For explanation of the use of abstraction in specification, see [Melham].

Mathematical Systems — Most formal specifications of computer architectures have been developed as part of the verification or synthesis of systems that include an architecture as an interface: processors, microcode, I/O devices, assembler code, compilers, and operating systems. These specifications are not generally of interest as architecture documentation, but are of interest in

demonstrating many of the mathematical systems that can be used to represent architectures.

A *state delta* expresses what changes an operation makes to a system's state. Assuming that some (explicitly stated) preconditions are met, the new state is defined in terms of the old state. [Marcus] introduces and justifies state deltas. [Crocker] demonstrates the use of state deltas for verifying microcode. State deltas model computation, and are used to describe the effects of the microcode; the target architecture itself is specified in ISP.

[Gordon83] introduces *LCF-LSM* (logic of computable functions/logic of sequential machines), and [Cullyer] demonstrates its use in specifying a processor. In more recent work, LCF-LSM has been superseded by higher-order logic.

[Dittmann] describes the use of *finite-state machines* and *formal grammars* for specifying architectures, and [Stoffel] for specifying I/O devices.

A specification with an *operational logic* is an example implementation. This has the advantages that it proves that the specification can be satisfied, and it provides a simulator for the architecture. With most operational specification languages, however, it is not possible to distinguish between state variables or events that are architecturally relevant from those that are artifacts of the example, and as a result, operational specifications tend to overspecify. It is also difficult to express constraints on parallelism, synchronization, and timing with an example implementation. [Zave] describes operational specification languages, [Bell] explains the need for a formal language, introduces an operational logic for specifying architectures (ISP), and [Parker] describes its limitations. [Clutterbuck] and [Falkoff] give operational specifications of architectures.

[Floyd] and [Hoare69] introduce a method for specifying an axiomatic semantics for programming languages. For each kind of program statement, a *Floyd-Hoare axiom* of the form {P}C{Q} defines its effects. P is called the axiom's precondition, and Q is called its postcondition. The axiom states that if the precondition is true, then after the statement C terminates, the postcondition is true. If there is a defining axiom for each type of statement, then it is possible to perform proofs to determine the effects of running a program. (Note that the axiom says nothing if C does not terminate. A proof with such axioms produces a statement of "partial correctness"—that is, correctness assuming that the program terminates. The method must be extended to prove "total correctness.") Floyd-Hoare axioms are known to have inherent limitations for specifying programming languages, as shown in [Clark]. [Damm] and [Dasgupta84] use Floyd-Hoare axioms in specifying architectures.

[Milner89] describes a formal system, which he calls a *process calculus*, for specifying and reasoning about systems of communicating parts (the calculus was previously called CCS [Milner80]). It has enough generality to describe architectures, yet no unnecessary power. [Milne] describes another early system.

An *algebraic specifications* defines a new sort of object (together with a set of operations) by means of axioms about the results of the operations. The operations defined for a sort are the only ones that can operate on objects of that sort. If the axioms are all equations (as opposed to implications, or inequalities, for example) then the specification is written in an *equational logic*, and can be considered a set of rewrite rules. If the rewrite rules meet some other conditions, any term can be automatically rewritten to a unique simplest form, which means that the specification can be executed. [Guttag75], [Ehrig], and [van Diepen] introduce algebraic specification of abstract data types, and [Sannella84] defines observational and behavioral equivalence for algebraically specified systems. Algebraic logics are mathematically elegant, but they can be very hard to read and write, even for experts. [Geser] argues that they need not be, and demonstrates with an architecture specification. [Frankel] demonstrates specification of a much simpler processor.

Functional (or applicative) programming languages express programs in which the output of each procedure is a simple function of its inputs—that is, side effects and internal state are forbidden. They are equivalent to equational specifications that can be executed. [Backus] argues for the use of functional languages for software, and [Streitz] argues for their use for architecture specification. [Sekar] demonstrates their use for architecture specification. [Hunt] finds a functional style makes combining (or dividing) specifications difficult.

[Reisig] introduces *Petri nets*, and [Prevost], [Li], and [Chiu] apply them to architecture specification. [Cohen86a] argues that they have several serious problems.

[Bjørner] introduces *VDM*, (the Vienna Definition Method), [Jones] and [Rumbaugh] use it to specify data-flow machines, and [Wichmann] uses it to specify floating point. VDM is a rigorous development method, not a formal system for representing specifications.

[Boyer79] defines a computational logic that has become known as *Boyer-Moore logic*. It was originally first-order predicate logic without quantifiers, and [Boyer88] adds quantifiers. [Hunt] uses the original logic to specify a processor, [Moore] specifies an assembler language, and [Bevier89] shows how to specify a parameterized architecture. Though the proof tool is powerful, the logic itself is limited, and this causes difficulties in specifying architectures, as described by [Hunt].

Temporal logics are predicate logics extended with the operators "hereafter" and "eventually" (and sometimes others), and are used to describe constraints on future states in terms of current states. The same constraints can be expressed in predicate logics without the temporal operators, by making states functions of time, but the temporal logics are more compact and direct. *Interval-temporal logics* are temporal logics in which predicates are bounded in time at both ends. [Schwartz82] introduces temporal logics, and [Schwartz83] introduces interval

logics. [Chiu] and [Li] use Petri nets and interval logic to specify time constraints on a processor. [Moszkowski] demonstrates a variety of temporal logic specifications. [Cohen86a] compares a variety of logics (for the purpose of specifying communication protocols), and finds temporal and interval logics best.

[Sufrin] introduces *Z*, which is a notation for set theory. [Bowen86,Bowen87] uses it to specify the Motorola 6800, [Bowen89,Farr] specify the inmos transputer, [Rose] specifies part of the Motorola 68000, and [Barrett] specifies floating point.

While first-order logic includes constants, and variables that represent constants, *higher-order logic* (also called *type theory*) includes variables that represent functions of constants, and functions of functions. The quantifier λ (lambda) is used to produce new functions, as in Lisp. [Hanna] argues for and demonstrates the use of higher-order logic in specification, and [Gordon88] introduces a mechanized version called HOL. [Cohn87,Cohn88] and [Joyce] apply it to processor verification.

Category theory is concerned with the relationships between different formal systems. An institution is much like what has been referred to here as a formal system; it is a signature (that is, some types that can be talked about), some sentences in that signature (some axioms), and a relation saying when another sentence in the signature satisfies the axioms (something like rules of inference). By mapping from one institution to another, category theory allows a specification in one formal system to be turned into a specification in another formal system. [ADJ] use category theory in specifying types, and [Goguen] and [Sannella85] argue that specifications should be considered independently of an underlying formal system, and show how this can be done with category theory, so that the formal systems used for specification become interchangable and the choice of formal system less important.

Many of these formal mathematical systems are objects of study themselves. Such study is particularly helpful in finding theoretical limits of expressiveness, in clarifying the restrictions necessary for soundness, and in identifying potentially useful extensions and generalizations. [Hatcher] and [Church] explain the logical foundations of mathematics, including completeness, consistency soundness, first-order logic, type theory (higher-order logics), set theories, intuitionism, and category theory. [Loomes] explains the mathematics of formal methods in computer science.

Annotated Bibliography

An Aside About Bibliography — After nearly completing this bibliography, I found three works explaining how annotated bibliography should be done. [Lambert] explains how to do a literature search in a scientific or technical field, and [Harner] and [Colaianne] explain how to write an annotated bibliography. They describe many of the difficulties, and suggest methods for avoiding them. In compiling this work, I had difficulties in two additional areas that they do not address.

First, what sorts of works should be cited? By the time I developed an acceptable criterion, I had collected three times as much literature as I eventually cited, but the criterion is so general that it could have been suggested by one of the books—the bibliography should include two kinds of works: everything that explicitly addresses the stated topic (in this case, architecture specification), and the best introductions to related fields. In practice, this bibliography does not include some works that explicitly address architecture description, because they merely repeat what is written in works that are cited. Further, it cites almost none of the many papers describing implementation specifications written in hardware-description languages, because there are a great many, and those I have examined are generally demonstrating a notation or tool rather than addressing issues of specification.

Second, what should the annotations say about the cited works? I now believe that each annotation should do three things. First, it should state the cited work's major points. If there are too many, as is common in textbooks and tutorials, then it should define the work's scope. Second, it should define the cited work's intended audience. That is, it should say what a reader must already know in order to understand the cited work. Third, it may note that the cited work is particularly well- or poorly written. The bibliographer should also say how the cited work is important in the context of the bibliography's topic, but I prefer to put that information in an associated survey, as is done here, rather than in the annotations. (I didn't write an explicit statement of what an annotation should say until after I had finished this bibliography, so the annotations herein are not all complete.)

Publishers' Addresses

Academic Press — Academic Press, Orlando, Florida 32887, USA.

ACM — Association for Computing Machinery, 11 West 42nd Street, New York, New York 10036, USA.

Addison-Wesley — Addison-Wesley Publishing, Reading, Massachusetts, USA.

AFIPS Press — AFIPS Press, 1815 North Lynn Street, Arlington, Virginia 22209, USA.

American Mathematical Society — American Mathematical Society, Providence, Rhode Island, USA.

Bonn — Universität Bonn, Institut für Informatik, West Germany.

Cambridge — Cambridge University Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, U.K.

Cambridge University Press — Cambridge University Press, The Edinburgh Building, Shaftesbury Road, Cambridge CB2 2RU, U.K.

Chicago — The University of Chicago Press, Chicago, Illinois 60637, USA.

CLinc — Computational Logic, Inc., 1717 West 6th Street, Suite 290, Austin, Texas 78703, USA.

Clive Bingley — Clive Bingley, 7 Ridgmount Street, London WC1E 7AE, U.K.

CMCS — Centrum voor Wiskunde en Informatica (Centre for Mathematics and Computer Science), P. O. Box 4097, 1009 AB Amsterdam, The Netherlands.

CMU — Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA.

Edinburgh — Department of Computer Science, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K.

Hong Kong — Department of Electrical Engineering, University of Hong Kong, Pokfulam Road, Hong Kong.

IBM — International Business Machines, Armonk, New York 10504, USA.

IEE — IEE, Savoy Place, London WC2R 0BL, U.K.

IEEE — IEEE, 345 East 47th Street, New York, New York 10017, USA.

Kluwer — Kluwer Academic Publishers, P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

Leeds — Centre for Theoretical Computer Science, Leeds University, Leeds LS2 9JT, U.K.

Longman — Longman Press, New York, New York, USA.

McGraw-Hill — McGraw-Hill, New York, New York, USA.

MacMillan — MacMillan Publishing, 866 3rd Avenue, New York, New York 10022, USA.

MIT — Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, USA.

MIT Press — MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, USA.

MLA — Modern Language Association, 10 Astor Place, New York, New York 10003, USA.

National Physical Laboratory — National Physical Laboratory, Teddington, Middlesex TW11 0LW, U.K.

Newcastle — Computing Laboratory, Claremont Tower, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, U.K.

North-Holland — North-Holland (a division of Elsevier Science Publishers), P.O. Box 211, 1000 AE Amsterdam, The Netherlands.

Oxford — Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, U.K.

Passau — Universität Passau, Fakultät für Mathematik und Informatik, Postfach 2540, D-8390 Passau, West Germany.

Pergamon — Pergamon Press, Headington Hill Hall, Oxford OX3 0BW, U.K.

Pitman — Pitman Publishing, 39 Parker Street, London WC2B 5PD, U.K.

Prentice-Hall — Prentice-Hall, Englewood Cliffs, New Jersey 07632, USA.

Princeton — Princeton University Press, Princeton, New Jersey, USA.

RSRE — Royal Signals and Radar Establishment, Malpern, U.K.

Springer-Verlag — Springer-Verlag, New York, New York, USA.

SRI — Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, California 94025, USA.

STC — Society for Technical Communication, 815 NW 15th Street, Washington D.C. 20005, USA.

Toronto — Department of Computer Science, 10 King's College Road, Room 3203, University of Toronto, Toronto, Ontario M5S 1A4, Canada.

UT Austin — Institute for Computing Science, University of Texas at Austin, Austin, Texas 78712, USA.

Wiley — John Wiley & Sons, New York, New York, USA.

Wredco — Wredco Press, Salt Lake City, Utah, USA.

Xerox PARC — Palo Alto Research Center, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, California 94304, USA.

References

Publishers' names are abbreviated in the bibliographic entries; the full names and addresses are listed in the previous section. The location of articles in serial publications is abbreviated as v(n):pp-pp. So, for example, 3(5):7-9 indicates volume 3, issue (or part) 5, pages 7 through 9.

ADJ **An initial algebra approach to the specification, correctness, and implementation of abstract data types.**
J. A. Goguen, J. W. Thatcher, and E. G. Wagner. IBM research report RC 6487, IBM, 1976. Also in Raymond T. Yeh, editor, *Current Trends in Programming Methodology, Volume 4: Data Structuring*, pages 80-149, Prentice-Hall, 1978.

Ardizzone **ISD: An instruction set descriptor for HDLs.**
E. Ardizzone and F. Sorbello. In *AICA 86 Annual Conference Proceedings*, pages 303-306. Associazione Italiana per l'Informatica ed il Colcolo Automatico, North-Holland, 1986.

Backus **Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.**
J. Backus. *Communications of the ACM*, 21(8):613-641, ACM, August 1978.

Barbacci79 **Specification, evaluation, and validation of computer architectures using instruction set processor descriptions.**
M. R. Barbacci, W. B. Dietz, and L. J. Szewerenko. In *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, pages 14-20, IEEE, 1979.

Describes the ISPS system for simulating computer architectures; explains some problems with specification of architectures (including incompleteness, inconsistency, and several kinds of overspecification); notes the desirability of being able to test the correctness of the specification; describes the comparision of formally described architectures using several metrics; and explains how formal specifications can be useful in verifying processor implementations.

Barbacci81 **Instruction set processor specification (ISPS): the notation and its applications.**
M. Barbacci. *IEEE Transactions on Computers*, 30(1):24-40, IEEE, January 1981.

Describes ISPS and its uses. The language is demonstrated by using it to specify the PDP-8 architecture and some pieces of implementation structure. ISPS descriptions have been used to document computer architectures; for quantitatively comparing computer architectures; as model implementations; for program verification by means of symbolic execution; for fault-insertion testing of diagnostics and fault-tolerant software; and as input to programs that generate architecture tests, implementation designs, assemblers, and compilers.

ISPS is an example of a very successful computer-hardware description language (CHDL,

or HDL). The author attributes some of its wide applicability to a provision for annotating an ISPS description with information (in the form of attribute-value pairs) that is actually to be used by other tools.

Barrett **Formal Methods Applied to a Floating Point Number System.**
Geoff Barrett. Technical monograph PRG-58, Oxford, January 1987.

Formally specifies IEEE Standard 754, for binary floating-point, in Z. Specifies the representation of floating-point numbers, their relationship to real numbers, conversions between various representations, and describes how arithmetic operations can be implemented correctly. This is part of a larger effort to verify the microcode of the floating-point unit of the inmos T800 transputer.

The specifications and theorems described are a start toward a useful theory of floating-point numbers. A more complete theory would contain most of the theorems needed for specification and verification of a wide range of computer implementations of floating-point arithmetic.

Bell **The PMS and ISP descriptive systems for computer structures.**
C. Gordon Bell and Allen Newell. In *Proceedings of the Spring Joint Computer Conference, 1970.* AFIPS Press, 1970.

In gathering material for the 1971 edition of [Siewiorek], Bell and Newell found that the original literature used very diverse, and often very poor, descriptive techniques, so they developed their own notations for rewriting the descriptions. The resulting languages are ISP and PMS. ISP is intended to be sufficient to describe the complete interface between computer and program, but is inadequate for describing the kinds of constraints on memory reference interference and ordering that can appear in the architecture of multiprocessors.

Berztiss **Specification and implementation of abstract data types.**
Alfs T. Berztiss and Satish Thatte. In *Advances in Computers*, 22:296-353. Academic Press, 1983.

Includes sections on the motivation for, nature of, and history of data abstraction; operational and algebraic specification; consistency and completeness; implementation and verification; and bibliography. Asserts that the benefit of data abstraction is that it collects all operations that read or modify a data structure into a textually and conceptually simple object, so the structure and what happens to it can be understood and proven. If other operations can read or write the structure, its effect can't be understood.

Bevier87 **Toward verified execution environments.**
William R. Bevier, Warren A. Hunt, Jr., and William D. Young. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 106-115. IEEE, 1987. Also technical report 5, CLinc, February 1987.

Explains how hierarchical verification can make a system more worthy of trust, and describes the verification of several layers of a system, including processor, compiler, and small multitasking operating system.

Bevier89 **A method for the formal specification of a class of instruction set architectures.**
William R. Bevier. Unpublished technical report, CLinc, 1989.

Uses a new feature of the Boyer-Moore logic, called functional instantiation, to express axioms about an architecture. The example axioms first assert the existence of the architectural state variables; then constrain their values (which may depend on the previous state); and finally constrain the effects of executing an instruction on an allowed state. In order to guarantee that the logic stays consistent, the axiom-definition procedure requires (as one of its arguments) a witness function to act as an existence proof, showing that the axiom can be satisfied.

The axioms defining an architecture need not totally define it, so such things as the word length and the memory size can be left unconstrained. The specification can thus allow a range of particular architectures, differing in those ways.

Bhandarkar **Architecture management for ensuring software compatability in the VAX family of computers.**
Dileep Bhandarkar. IEEE *Computer*, 15(2):87-93, IEEE, February 1982.

Describes the need for managing an architecture and how the VAX architecture [Leonard] is managed. Points out that the specification must be precise; that a specification of a complex system will have defects; that an architecture and its specification change over time; that a specification should specify results, not implementations; that an architecture need not be complete (that is, it may leave some visible aspects of behavior undefined); and that implementations must be verified. Briefly describes the VAX architecture-management process and identifies some attributes essential to its success.

Bjørner **The Vienna Development Method.**
Dines Bjørner and Cliff Jones. Lecture Notes in Computer Science, 61, Springer-Verlag, 1978.

Bowen86 **The Formal Specification of a Microprocessor Instruction Set.**
Jonathan Bowen. Technical monograph PRG-60, Oxford, January 1986.

Specifies the instruction set of the Motorola 6800 microprocessor, including interrupts and memory, using the formal language Z. The specification structure and the language are intended to make the specifications easy to read, and they do much better than most formal languages. For example, the specification 'schemas,' the descriptions of addressing modes, the instruction formats, and the instruction sub-types are not strictly necessary, but correspond to the way engineers think about architectures, and greatly simplify the specification.

As presented, the specification style has three minor deficiencies. First, memory is modelled as a function from addresses to values, which doesn't extend well to multiprocessors or systems with direct-memory-access (DMA) I/O devices. Second, the specification does not seem to distinguish architecturally visible state and events from state and events that are merely expositorally useful. The need for the distinction is easy to overlook because the specification uses only architecturally visible state, and does not formally define the sense in which an implementation can satisfy the specification.

Third, Z is based on set theory, a formalism unfamiliar to many readers of architecture specifications. It's not clear that there's a better alternative, however.

Even with its deficiencies, this is the by far the most readable formal architecture specification, and the most formal readable architecture specification I've seen.

Bowen87 **Formal specification and documentation of microprocessor instruction sets.**
Jonathan P. Bowen. In H. Schumny and J. Mølggard, editors, Microcomputers: Usage, Methods, and Structures, *Microprocessors and Microprogramming, Proceedings of the 13th Euromicro Symposium*, 21:223-230, North-Holland, 1987.

Argues that computer-architecture specifications can and should be written using a readable formal notation. Notes that while informal architecture specifications are easy to read, they are often incomplete or ambiguous, and while formal specifications allow formal reasoning with the specification, they are hard to read. Presents the notation used in [Bowen86] as a formalism that is readable enough to be used as primary documentation.

Bowen89 **Z Specification of the ProCoS Level 0 Instruction Set.**
Jonathan Bowen and Paritosh Pandya. To appear in the ProCoS Workshop, Oxford, November, 1989.

Specifies a subset of the instruction set of the inmos transputer. Defines computer arithmetic, processor state, instructions, power-up and bootstrapping, and ends with a glossary of Z notation. The specification has been type-checked but not verified. It is derived from the more complete [Farr].

The specification assumes familiarity with Z, and is presumably intended to support formal reasoning about a compiler or code written in assembler for the μtransputer, and is not intended as primary documentation of the processor.

Boyer79 **A Computational Logic.**
Robert S. Boyer and J Strother Moore. ACM Monograph Series, Academic Press, 1979. Also technical report 55, UT Austin, 1978.

Defines the Boyer-Moore logic and is the user's manual for the theorem prover. Boyer-Moore logic is first-order predicate calculus without quantifiers.

Boyer88 **The addition of bounded quantification and partial functions to a computational logic and its theorem prover.**
Robert S. Boyer and J Strother Moore. *Journal of Automated Reasoning*, 4(2):117-172, Kluwer, June 1988.

Browning **Guide to Effective Software Technical Writing.**
Christine Browning. Prentice-Hall, 1984.

Carlson **Annotated Bibliography on Technical Writing, Editing, Graphics, and Publishing.**
Helen V. Carlson et al. STC, 1983.

Case **Architecture of the IBM System/370.**
R. P. Case and A. Padegs. *Communications of the ACM*, 21(1):73-96, ACM, January 1978.

Reviews the history and significance of the System/370 family. Includes a section describing IBM's architecture control procedure, which notes the need for a single, clear, and unambiguous description of the architecture.

Chen **Modeling of the 6809 through the hardware description languages.**
W. J. Chen and G. N. Reddy. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 60-68, IEEE, 1987.

Chicago **The Chicago Manual of Style.**
Chicago, 1982.

If you're intending to write a book to be published, you need this book or [Skillin]. Can be read as an introduction to bookmaking, but is primarily a reference work for author and editor. Describes the parts of a book, manuscript preparation, editing, production, and printing. Covers issues of style in extensive detail; from spelling and displaying foreign languages to how to caption illustrations and how to format bibliographies.

Chiu **Interval Logic and Modified Labelled Net for System Specification and Verification.**
P. P. K. Chiu. Master's thesis, Hong Kong, 1985.

Specifies and verifies a microcomputer implementation using interval logic and Petri nets. See also [Li] **A Higher Order Language for Describing Microprogrammed Computers.**

Church **Introduction to Mathematical Logic.**
Alonzo Church. Princeton, 1970.

Introduces formal logic, propositional calculus, higher-order logics, set theory, intuitionism, and more.

Clark **The characterization problem for Hoare logics.**
E. M. Clark, Jr. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, Prentice-Hall, 1985.

Clutterbuck **The verification of low level code.**
D. L. Clutterbuck and B. A. Carré. *Software Engineering Journal*, 3(3):97-111, IEE, May 1988.

Demonstrates the automatic translation of (a restricted subset of) 8080 assembler code into a program-modelling language, and the proof of verification conditions for programs written in assembler. The model of the 8080 is expressed in FDL, which has an operational style, and is evidently given a Floyd-Hoare style semantics, but the semantics are not described in the paper.

Cohen86a **The Importance of Time in the Specification of OSI Protocols: An Overview and Brief Survey of the Formalisms.**
B. Cohen, D. H. Pitt, and J. C. P. Woodcock. Technical report DITC 78/86, National Physical Laboratory, 1986.

Discusses the problems of specifying time-related behavior. Gives examples of time-related behavior in network protocols, summarizes and critiques the literature describing several techniques for specifying time-related behavior, and discusses difficulties of systems without global time, and systems in which specified global times can be implemented in terms of local times. When considering particular logics, notes that timing can be specified by relative orderings of events and that the technique can be used with any state-based formalism, but notes that there are open questions about how implementations can be verified; argues that real-time attribute grammars are more appropriate to fast prototyping and testing than to performance analysis or formal verification; argues that concurrent state deltas are inappropriate for distributed systems; argues that timed Petri nets (and a restricted class called coupled time graphs) have serious problems; and recommends that temporal or interval logics be used for specifying OSI protocols.

Cohen86b **Specification of Complex Systems.**
B. Cohen, W. T. Harwood, and M. I. Jackson. Addison-Wesley, 1986.

Cohn87 **A Proof of Correctness of the Viper Microprocessor: The First Level.**
Technical report 104, Cambridge, January 1987.

Describes the proof of equivalence of the top two levels of formal description of the Viper microprocessor, using higher-order logic. The top-level specification is the functional specification of the instructions, and the second-level specification is the major-state model.

Cohn88 **A proof of correctness of the Viper block model: the second level.**
Avra Cohn. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1-91, Springer-Verlag, 1989. Also technical report 134, Cambridge, May 1988.

Describes the partially completed specification and proof of the microcode-level model of the Viper microprocessor.

The top-level specification of Viper ignores the effects of resetting the chip, memory time-out, single-step, externally forced errors, and more. Cohn suggests exploring better (by which I assume she means more complete) top-level specification, and clarifies the limited assurances that formal verification gives.

Cohn89 **The notion of proof in hardware verification.**
A. J. Cohn. *Journal of Automated Reasoning*, 5:127-139, Kluwer, May 1989.

Discusses what it means to say that a system is verified. Makes five major points: neither an intended behavior nor a physical chip (a realization) can be proven, only models of them can be proven, and the models may not adequately represent either intentions or realization; in practice, the verified design and the manufactured design may be different; the verification is limited to the levels of detail that the specifications

cover, so a system is only verified between particular levels of detail; a verification makes assumptions about initial states and the normalcy of environmental conditions, and is valid only under those circumstances; and the verified design is typically only part of a system, and correct functioning of the system depends on much more, from the system operators to the mechanical parts. Does not discuss the possibility that the formal proof itself may be invalid. Avoiding this problem is a reason for using well-understood mathematical systems rather than ad hoc logics, and for mechanical proof-checking.

Colaianne **The aims and methods of annotated bibliography.**
A. M. Colaianne. *Scholarly Publishing*, 11(4):321-331, Toronto, July 1980.

Intended for prospective authors of scholarly bibliographies in the humanities. Opens by defending the art of bibliography from criticisms that it is essentially non-creative and mechanical, then provides advice about writing one. Assumes that the author has determined exactly what field is to be covered. Says to choose a style (checklist, digest, or analysis); decide which media to include (theses, books, articles, etc.); emphasizes the need for care in citation; warns against grouping citations arbitrarily or too finely; suggests that annotation avoid value judgements, capture the arguments of the cited work, and use a consistent and easy-to-read style; and suggests that the bibliography include a section describing the criteria for choosing works to cite, and a section of insights the author gained while reviewing the cited works as a group.

Crocker **Reverification of a microprocessor.**
Stephen D. Crocker, Eve Cohen, Sue Landauer, and Hilarie Orman. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 166-176, IEEE, April 1988.

Describes the verification of the FM8501 using SDVS, and compares the use of state deltas (in SDVS) to the use of first-order predicate logic (in Hunt's verification using the Boyer-Moore theorem prover). The authors mention that they've also specified and verified the microcode of the BBN C/30, but that the results haven't been published.

Cullyer **Application of formal methods to the VIPER microprocessor.**
W. J. Cullyer and G. H. Pygott. In *IEE Proceedings, Part E, Computers and Digital Techniques*, 134(3):133-141, IEE, May 1987,

Describes the Viper microprocessor and the specification methods used to verify the correctness of its top-level design. The top-level specification (in LCF-LSM) appears as an appendix.

Damm **A microprogramming logic.**
Werner Damm. *IEEE Transactions on Software Engineering*, 14(5):559-574, IEEE, May 1988,

Describes a method for verifying microcode written in a high-level, architecture-dependent programming language (S_A^*). The target and host architectures are each specified in AADL [Dasgupta84], and a Floyd-Hoare semantics for S_A^* is generated from the host AADL specification, and can be used to prove properties of the program.

Darringer **The Description, Simulation, and Automatic Implementation of Digital Computer Processors.**
J. A. Darringer. PhD thesis, School of Electrical Engineering and Systems and Communications Sciences, CMU, May 1969.

Dasgupta82 **Computer design and description languages.**
Subrata Dasgupta. In M. C. Yovits, editor, *Advances in Computers*, Academic Press, 1982.

Describes the need for computer hardware description languages, describes some of their differentiating features (in particular abstraction level and operational versus functional style), and then describes several languages (ISPS, SLIDE, ADL, S_A^*, and CONLAN).

Dasgupta84 **The Design and Description of Computer Architectures.**
Subrata Dasgupta. Wiley, 1984.

Defines computer architecture; distinguishes between informal and formal design processes (formal processes use formal descriptions of the object being designed); considers attributes of a good architecture description language; and develops a language (AADL) and tools for describing architectures and compiling descriptions into microcode.

AADL is an outgrowth of computer-hardware description languages, and, like most of them, looks and acts like a programming language, but is unusual in that it is given a formal (Floyd-Hoare) semantics. The tools allow an architecture description written in his language to be compiled into microcode for a lower-level architecture for which there is also an axiomatic specification. Dasgupta considers several attributes of description languages: level of abstraction; operational vs. functional; procedural vs. nonprocedural; the ability to specify behaviors vs. structures; and the influences of programming languages. By the level of abstraction, he means whether the language includes operators that directly model hardware components. He considers operational and functional models, but ignores all other kinds of semantics. The distinction between behaviors or structures is confused, failing to distinguish between an interface, which is necessarily a behavioral specification, and an implementation (a mapping between two interfaces) which is necessarily a structural specification. The confusion results from the common (and perfectly valid) practice of specifying behavior by abstracting from an implementation.

Dasgupta85 **On the axiomatic specification of computer architectures.**
Subrata Dasgupta and J. Heinanen. In *Proceedings of the 7th International Conference on Hardware Description Languages*, IEEE, 1985.

David **Architecture language (computer system description).**
G. David. *Tanulmanyok Magyar Tudomanyos Akademia Szamitastechnik es Automatizalosi Kutato Intezete*, 100:341-349, 1979. Hungary.

van Diepen **Implementation of Modular Algebraic Specifications.**
N. W. P. van Diepen. Report CS-R8801, CMCS, 1989.

Briefly but clearly introduces algebraic specification, modular algebraic specification, and Floyd-Hoare axioms; introduces observation functions to abstract from an algebraic specification; and argues that it's important to use observation functions to abstract

from a specification so that implementations can be optimized, and that this advantage offsets the loss of initiality (a formal property that is considered important in algebraic specification).

Dijkstra **A Discipline of Programming.**
E. W. Dijkstra. Prentice-Hall, 1976.

Dittmann **Finite-state machines and formal grammars as means of hardware description for computer architectures.**
J. Dittmann. *Siemens Forsch Entwicklungsber*, 9(5):294-297, Springer-Verlag, 1980.

Uses finite-state machines and formal grammars to specify transitions between five architectural states (powered off, stopped, user mode, executive mode, and error).

Ehrig **Fundamentals of Algebraic Specification I: Equations and Initial Semantics.**
H. Ehrig and B. Mahr. EATCS (European Association for Theoretical Computer Science) Monograph, 6, Springer-Verlag, 1985.

Eichenseher **CADL — A formal description language for parallel computer architectures.**
Ingo Eichenseher, Theo Ungerer, and Eberhard Zehendner. In *Microprocessing and Microprogramming*, The Euromicro Journal, 24(1-5):363-370, North-Holland, August 1988.

Introduces CADL, a language for specifying computer architectures. By architecture, the authors mean the overall structure of a design, so this is a computer-hardware description language intended for the top levels of a processor design. It is not given a formal semantics.

Falkoff **A Formal Description of System/360.**
A. Falkoff, K. Iverson, and E. Sussenguth. *IBM Systems Journal*, 3(3):198-262, IBM, 1964.

Describes the architecture of the System/360 in APL, including all processor state, console, memory, interrupts, and I/O. Console and I/O operations can be taken to be the architecture's observation functions. Thus, although it is an operational specification (an implementation), the observation functions can be used to abstract from it to produce an interface specification. Unfortunately, the specification is extraördinarily difficult to read.

IBM does not use APL for architecture specifications because they feel that the language is not flexible enough to express all they needed to say, and because the language can't be understood and used by the intended audience without training. See [Gifford].

Farr **A Formal Specification of the Transputer Instruction Set.**
J. R. Farr. Master's thesis, Oxford, 1987.

Floyd **Assigning meanings to programs.**
R. W. Floyd. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19-32, American Mathematical Society, 1967.

Frankel **Beyond register transfer: an algebraic approach for architectural description.** R. E. Frankel and S. W. Smoliar. In *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, pages 1-5, IEEE, 1979.

Argues that architecture descriptions should not be based on register-transfer descriptions, and that abstract data types provide a better style. Asserts that register-transfer operations have shortcomings because they are equivalent to assignment statements (but doesn't make the shortcomings of assignment statements clear), and then demonstrates the specification of a simple processor as an abstract data type.

Geser **A Specification of the intel 8085 Microprocessor: A Case Study.** Alfons Geser. In M. Wirsing and J. A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, Lecture Notes in Computer Science, 394:347-402, Springer-Verlag, 1989. This is a revised version of technical report MIP-8608, Passau, May 1986.

Argues that it is possible to write large algebraic specifications in such a way that they are understandable and maintainable, and demonstrates by specifying the interface to the intel 8085 processor. Assumes familiarity with algebraic specification and rewriting, but introduces the specification languages used (RAP and COLD-K). Defines bits, bytes, and words; processor state (several registers); three abstract buses through which the processor communicates with its environment; operations on those buses; the effects of several example instructions; and the effects of the instruction-execution cycle; and tests the specification by executing it on test cases and simple proofs. The full specification is included as an appendix.

Gifford **Case study: IBM's System/360-370 architecture.** David Gifford and Alfred Spector. *Communications of the ACM*, 30(4):292-307, ACM, April 1987.

The authors interview Andris Padegs and Richard Case of IBM about the architecture's scope, evolution, future, description, maintenance, and control. Mentions that IBM does not use the APL description as the architecture specification because they feel that the language is not flexible enough to express all they need to say, and because the language can't be understood and used by the intended audience without training.

Goguen **Introducing institutions.** J. A. Goguen and R. M. Burstall. In E. Clarke and D. Kozen, editors, *Logics of Programs*, Lecture Notes in Computer Science, 164:221-256, Springer-Verlag, 1984.

Gordon83 **LCF-LSM, A System for Specifying and Verifying Hardware.** M. J. C. Gordon. Technical report 41, Cambridge, September 1983.

Gordon88 **HOL: A Proof Generating System for Higher-Order Logic.** Mike Gordon. Kluwer, 1988. Also technical report 103, Cambridge, January 1987.

Describes the HOL system and logic. The HOL system is described much more thoroughly in its documentation, which is available through Mike Gordon at Cambridge.

Gries **Programming Methodology — A Collection of Articles by Members of IFIP WG2.3.**
D. Gries, editor. Springer-Verlag, 1978.

Guttag75 **The Specification and Application to Programming of Abstract Data Types.**
J. V. Guttag. PhD thesis, report CSRG-59, Toronto, 1975.

Guttag80 **Formal specification as a design tool.**
J. V. Guttag and J. J. Horning. In *Proceedings of the Principles of Programming Languages Conference*, pages 251-261, ACM, 1980. Also technical report CSL-80-1, Xerox PARC, January 1980.

Argues that the major benefit of formal specification is clearer understanding of the specified design; proposes that the abstract objects and operations be specified separately from the routines that use them or implement them; and argues for and demonstrates the testing of formal specification by using proof to check the answers to informal questions about the intended design.

Haney **Using a Computer to Design Computer Instruction Sets.**
Frederick M. Haney. PhD thesis, Department of Computer Science, CMU, May, 1968.

Hanna **Specification and verification of systems using higher-order predicate logic.**
F. K. Hanna and N. Daeche. In *IEE Proceedings, Part E, Computers and Digital Techniques*, 133(5):242-254, IEE, September 1986.

An exceptionally lucid paper, introducing much of the framework of the field, including the meaning and justification of formalisms; how higher-order logic can be used to specify system behavior and system structure; and how specifications can be used for verification.

Argues that higher-order logic is a good formalism because it allows partial specifications of behavior; allows specification of behavior, structure, low-level timing, and hierarchies of behavior, structure, and timing; uses the well-researched deductive calculus of predicate logic, so the pitfalls of logical paradoxes are well mapped and avoidable; is sufficiently powerful to concisely describe very complex notions; and allows specifications and deductions to be structured into theories. Distinguishes between a formal theory and an interpretation; gives examples of higher-order logic axioms, including a partial axiomitization of time and of analog waveforms; illustrates the specification of digital behaviors and structures; describes mechanical theorem proving and verification, and explains how and why mechanical theorem proving is used with formal specifications.

Harman **The Formal Specification of a Digital Correlator I: User Specification Process.**
N. A. Harman and J. V. Tucker. Report 9.89, Leeds, 1989.

Develops a model of the process of developing specifications; develops a mathematical model for specifying functions that tranform infinite signal streams; then takes a digital correlator through the modelled process of design, producing ten specifications by stepwise refinement. Introduces the literature of digital systems design, design theory, verification and synthesis, and complexity theory.

Among other things, the authors argue that the result of design should not be just an implementation, but must also include a formal specification, a description of how the formal specification models the informal specification, and tests and proofs to show that the implementation meets the formal specification (in function and performance) and the formal specification meets the informal one.

Harner **On Compiling an Annotated Bibliography.**
James L. Harner. MLA, 1985.

A very helpful guide, especially to the first-time compiler of a bibliography. Covers the purpose, organization, and process of compiling any bibliography, but especially an annotated one.

Hatcher **The Logical Foundations of Mathematics.**
William S. Hatcher. Pergamon, 1982.

Defines mathematics and formal systems, explains how they depend on logical argument, how that dependence has led to study of the logic used in mathematical argument, and how mathematical systems have been redefined in terms of logic. Explains several of the redefinitions, the history of their development, and the current state of understanding.

This book clearly explains why there is a choice of formal systems, and why the choice matters. It defines such terms as formal language, formal system, deductive calculus, model, and so on.

Hoare69 **An axiomatic basis for computer programming.**
C. A. R. Hoare. *Communications of the ACM*, 12(10):576-583, ACM, October 1969.

Proposes making axioms and rules of inference explicit for programming-language constructs, to support program proof and language definition, and demonstrates it on tiny examples.

Hoare85 **Communicating Sequential Processes.**
C. A. R. Hoare. Prentice-Hall, 1985.

Horning **Some notes on putting formal specifications to productive use.**
J. J. Horning. In M. J. Elphick, editor, *Formal Specification, Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar*, pages 117-132, Newcastle, 1983. Also John Guttag, Jim Horning, and Jeannette Wing, technical report CSL-82-3, Xerox PARC, June 1982.

Gleanings from the authors' experiences in writing formal specifications: the clarity produced in writing a formal specification is often more valuable than the specification itself; specifications should be written as the decisions they record are made, but even after-the-fact specification can be useful; specifications should be written by the designers; and specifiers may want to use different languages to specify different kinds of constraints in a single specification. Lists some software tools the authors want (syntax and type checker, theorem prover, library of specifications, editor, and viewer).

Hunt **FM8501: A Verified Microprocessor.**
Warren Hunt, Jr. Technical report 47, UT Austin, December 1985.

Specifies a microprocessor (called the FM8501) at both the gate level and the instruction-behavior level, then proves that the gates implement the instruction behavior. Uses Boyer-Moore logic and the Boyer-Moore theorem prover. The behavioral specification is operational in style.

Hunt found the lack of quantification in the logic to be a problem when representing nondeterminism (which showed up in his specification in the clocking of an external memory). He also found that the functional style made composition of separate specifications difficult. It will, for example, make it difficult to specify systems in which several processors or I/O devices access a shared memory.

IBM **IBM System/370 Principles of Operation.**
International Business Machines. Order number GA22-7000, IBM.

inmos **The Transputer Instruction Set: A compiler writer's guide.**
inmos Limited. Prentice-Hall, 1988.

Iverson **A Programming Language.**
K. E. Iverson. Wiley, 1962.

Describes the APL notation and illustrates its use in a variety of applications. Chapter 2 illustrates its use as an architecture specification language by describing the IBM 7090 at the register-transfer level.

Iverson's emphasis is on APL as a notation, and he achieves a very compact notation. APL's strengths add no power as a logic, however, and its conciseness is more than made up for by its obscurity. As a result, APL has not been a success as a specification language.

Jones **A formal semantics for a dataflow machine–using VDM.**
K. Jones. In D. Bjørner and C. Jones, editors, *VDM '87: VDM–A Formal Method at Work*, Proceedings of the VDM-Europe Symposium, Lecture Notes in Computer Science, 252, Springer-Verlag, 1987.

Joyce **Multi-Level Verification of Microprocessor-Based Systems.**
Jeffrey John Joyce. PhD dissertation, Cambridge, December 1989.

Demonstrates the use of higher-order logic to verify several connected layers in a computer system, from programming language to register- transfer level. Argues for the use of generic specifications to hide detail irrelevant to each layer; argues for embedding familiar notations in higher-order logic and using them in the familiar ways; and argues that higher-order logic is a particularly good formalism.

Kershaw **Viper: A Microprocessor for Safety-Critical Applications.**
J. Kershaw. RSRE Memo 3754, RSRE, 1985.

Kuehne **Handbook of Computer Documentation Standards.**
Robert S. Kuehne, Herbert W. Lindberg, and William F. Baron. Prentice-Hall, 1973.

Lambert **How to Find Information in Science and Technology.**
Jill Lambert and Peter A. Lambert. Clive Bingley, 1986.

Explains how to do a literature search. Describes channels through which information is distributed (journals, conferences, reports, etc.) and how the information varies across channels; how to use libraries, various types of catalogues, lists of published books, reviews, theses, abstracts and indexes; what information to record from each citation, and why; on-line databases and how to search them; how to obtain literature, given a citation; how to organize and maintain a collection of citations; how to stay current after a search; and closes with some speculations about how literature might be published in the new channel of computer networks.

If you have not done a thorough literature search before, this should be quite a useful introduction. The presentation isn't as high-quality as one might wish, but the contents are relevant, useful, and sufficiently thorough to be well worth the trouble.

Leonard **VAX Architecture Reference Manual.**
Timothy E. Leonard. Digital Press, Bedford, Massachusetts, 1987.

Informally specifies the VAX architecture, using English, occasional pieces of pseudo-Algol code, and diagrams. Specifies instruction formats, addressing modes, instruction execution, memory management, exceptions and interrupts, process-context switching instructions, shared memory, bootstrapping, and the console.

Levy **Microcode verification using SDVS: the method and a case study.**
Beth Levy. In *Proceedings of the 17th Annual Microprogramming Workshop*, printed as a special issue of the ACM SIGMICRO Newsletter, 15(4):234-245, ACM, December 1984.

Describes SDVS (state-delta verification system), explains how it can be used to verify microcode, and provides an example. The target architecture is described in ISPS.

Li **Microcomputer system specification using interval logic and a modified labelled-net model.**
H. F. Li, Y. S. Cheung, and P. P. K. Chiu. In *IEE Proceedings, Part E, Computers and Digital Techniques*, 133(4):223-234, IEE, July 1986.

Largely concerned with the specification of timing constraints and the verification of an interval-logic specification by means of Petri nets. Though the microcomputer specification is referred to, only a small part is illustrated. [Chiu] provides the complete example.

Liskov **Abstraction and Specification in Programming Development.**
B. Liskov and J. Guttag. MIT Press, 1986.

A textbook for undergraduates with some programming experience. Uses the language CLU to teach the importance and techniques of abstraction and specification.

Loomes **Software Engineering Mathematics: Formal Methods Demystified.**
M. Loomes and J. C. P. Woodcock. Pitman, 1988.

Marcus **SDVS: a system for verifying microprogram correctness.**
Leo Marcus, Stephen D. Crocker, and Jaisook R. Landauer. In *Proceedings of the 17th Annual Microprogramming Workshop*, printed as a special issue of the ACM SIGMICRO Newsletter, 15(4):246-255, ACM, December 1984.

Justifies microcode verification, introduces state deltas, describes SDVS (state-delta verification system), and summarizes the state of the SDVS project. SDVS uses ISPS as its architecture-specification language.

Melham **Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic.**
Thomas Frederick Melham. PhD thesis, Cambridge, 1989.

Describes formal abstraction mechanisms for hardware specification and verification; describes behavioral, data, and temporal abstraction; and describes abstraction of the model of a class of devices.

Melliar79 **System specification.**
P. M. Melliar-Smith. In T. Anderson and B. Randell, editors, *Computing Systems Reliability*, pages 19-65, Cambridge University Press, 1979.

Discusses system specification, particularly formal specification of computer programs, and gives a good view of the state of the art at the time of writing. Starts by justifying specification and formal specification; describes the process of going from a specification to an implementation using an example design process; introduces the mathematics of specification, including axiomatic and operational semantics, abstract and representational types, abstract algebras and heterogenous algebras, rewrite rules, and an algebraic method of specifying types; introduces and demonstrates the specification language Special; discusses incomplete specifications, especially asynchronous systems, including deterministic, serializable, and unserializable systems, and temporal logic. Notes that operational specifications can have the advantages of familiarity, understandability, and inherent consistency and implementability, but can have the disadvantages of overspecification and greater difficulty proving that a different implementation meets the specification. Algebraic specifications have their own difficulties, especially with operations that return more than a single value or that may return an error, and they are difficult to read, and difficult to write even for an expert.

Melliar82 **Formal specification and mechanical verification of SIFT: a fault-tolerant flight control system.**
P. M. Melliar-Smith and R. L. Schwartz. *IEEE Transactions on Computers*, C-31(7):616-630, IEEE, July 1982, Also technical report CSL-133, SRI, 1982.

Describes the specification and proof methods used on SIFT, a highly reliable computer for aircraft control. Describes the software-voting scheme used for reliability; explains how a hierarchy of proofs link the Pascal code and the hardware fault model to the high-level reliability rates; and gives examples from the hierarchy of specifications. The compiler and hardware are not specified or verified. Notes that the correspondence between formal specification and intentions must be determined by inspection, so the top-level specification must be believable.

Meyer **On formalism in specifications.**
B. Meyer. In *IEEE Software*, 2(1):6-26, IEEE, January 1985,

Milne **A Mathematical Model of Concurrent Computation.**
G. J. Milne. Technical report CST-478, PhD thesis, Edinburgh, 1978.

Milner80 **A Calculus of Communicating Systems.**
Robin Milner. Lecture Notes in Computer Science, 92, Springer-Verlag, 1980.

Introduces CCS. Superseded by [Milner89].

Milner89 **Communication and Concurrency.**
Robin Milner. International Series in Computer Science, Prentice-Hall, 1989.

Introduces a process calculus for describing systems with communicating parts; provides an underlying theory and examples of use.

The theory described by this book is elegantly small, and powerful enough to describe many aspects of computer systems that are of interest. The book clarifies many issues of specification and proof of concurrent systems.

Moore **A Mechanically Verified Language Implementation.**
J Strother Moore. Technical report 30, CLinc, September 1988.

Describes the language Piton, an implementation of a Piton compiler and link-loader, and a proof of the implementation's correctness. The Piton formal specification is an operational specification in the form of an interpreter written in the Boyer-Moore logic.

Piton is a high-level assembly language, and provides execute-only programs, recursive subroutine call and return, stack-based parameter passing, local variable, global variables and arrays, a user-visible stack for intermediate results, and seven abstract data types.

Moore notes several subtle issues with mapping the implementation results into top-level results. A verifiable implementation of any specification must include not only a lower-level specification, but also the mapping between objects and events visible at the top level to objects and events at the lower level, something many specifiers overlook.

Mosenthal **Research on Writing: Principles and Methods.**
Peter Mosenthal, Lynne Tamor, and Sean A. Walmsley, editors. Longman, 1983.

Moszkowski **Executing Temporal Logic Programs.**
B. Moszkowski. Cambridge University Press, 1986.

Padegs *See* [Gifford] *and* [Case].

Parker **ISPS: a retrospective view.**
Alice C. Parker, Donald E. Thomas, Stephen Crocker, and Roderic G. G. Cattell. In *Proceedings of the 4th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 21-27, IEEE, 1979.

Describes the originally intended purpose of ISPS and the problems that users of the language have discovered. The problems described include inadequate support for

concurrency, timing, or synchronization; lack of some commonly used operations as primitives; and difficulty in separating some parts of a specification—for example, the description of address computation from instruction execution. Requirements for an ideal behavior-description language include an abstraction facility, specification of behavior without overconstraining implementation, the ability to specify synchronization primitives directly, and formal semantic definition of the language.

Parnas **On the criteria to be used in decomposing systems into modules.**
David. L. Parnas. *Communications of the ACM*, 15(12):1053-1058, ACM, December 1972.

The author's conclusion: "We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing."

Prevost **A methodology for verification of complex architectures by Petri-nets.**
G. Prevost and M. Currat. In *ESPRIT '86: Results and Achievements*, pages 491-502, North-Holland, 1987.

Reisig **Petri Nets: an Introduction.**
W. Reisig. EATCS (European Association for Theoretical Computer Science) Monograph, Springer-Verlag, 1985.

Rose **A Partial Specification of the MC68000 Microprocessor.**
Phillip B. Rose. Master's thesis, Oxford.

Rumbaugh **A Parallel Asynchronous Computer Architecture for Data Flow Programs.**
J. E. Rumbaugh. PhD thesis, MIT, May 1975.

Presents the design of a data-flow language and processor, and informally proves that the processor design implements the language. The designs are specified with the Vienna Definition Method, using a formal language which is not given a formal semantics.

Sannella84 **On observational equivalence and algebraic specification.**
Donald Sannella and Andrzej Tarlecki. Internal report CSR-172-84, Edinburgh, 1984. Extended abstract in *Mathematical Foundations of Software Development, volume I: Proceedings of the Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science, 185:308-322, Springer-Verlag, 1985.

Formally defines observational equivalence and behavioral equivalence in terms of algebraic specifications. Two specifications are behaviorally equivalent with respect to particular sorts if the specifications give the same results for all operations that produce those sorts. (Informally, that means that comparison is only allowed for particular data types.) Two specifications are observationally equivalent if they give the same results for all comparisons from a prespecified set of comparions. (Informally, that means that only particular observations are allowed.) Assumes the reader has a basic understanding of algebraic specification, bit it is quite clear for such a formal paper.

Sannella85 **Building Specifications in an Arbitrary Institution.**
Donald Sannella and Andrzej Tarlecki. Internal report CSR-184-85, Edinburgh, 1985. Also in *Proceedings, International Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science, 173:337-356, Springer-Verlag, 1985.

Schwartz82 **From state machines to temporal logic: Specification methods for protocol standards.**
R. L. Schwartz and P. M. Melliar-Smith. In C. Sunshine, editor, *Protocol Specification, Testing, and Verification*, North-Holland, 1982.

Schwartz83 **Interval logic: A higher-level temporal logic for protocol specification.**
R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing, and Verification III*, North-Holland, 1983.

Sekar **Formal verification of a microprocessor using equational techniques.**
R. C. Sekar and M. K. Srivas. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 171-217, Springer-Verlag, 1989.

Specifies and verifies a very simple microprocessor (it has two instructions: STORE PC and JUMP) using a functional language called SBL. The authors promote their use of an explicit *random* function to describe behavior that is left unconstrained by the architecture.

Shiva **On the description of multiprocessor architectures.**
S. G. Shiva. In *Proceedings of the Real-Time Systems Symposium*, pages 101-110, IEEE, 1981.

Presents a syntax for specifying multiprocessor structures in terms of processors, memories, switches, and links, which are taken as primitives.

Siewiorek **Computer Structures: Principles and Examples.**
D. P. Siewiorek, C. G. Bell, and A. Newell. McGraw-Hill, 1982.

A classic reference, listing some 250 computers and describing 40 in some detail, covering an extremely wide variety of structures. Includes reprints of many original descriptions, and many descriptions rewritten in ISP and PMS, notations that were developed for the book.

Skillin **Words into Type.**
Marjorie E. Skillin and Robert Malcolm Gay. Prentice-Hall, 1974.

Like *The Chicago Manual of Style*, an exceptionally good reference book for authors and editors.

Stephen **Writing User-Usable Manuals: A Practical Guide to Preparing User-Friendly Computer Hardware and Software Documentation.**
Peter M. Stephen. Wredco, 1984.

Stoffel **Peripheral devices: a formal description of their behaviour.**
C. Stoffel. *Elektron Rechenanlagen*, 21(3):113-119, June 1979.

(In German.) The author proposes the use of automata rather than algorithmic languages to describe the operation of peripherals (demonstrating the technique by specifying a magnetic-tape controller), and then discusses desirable qualities of peripheral-device architectures in general.

Streitz *Rechnerspezifikation mit Hilfe einer Functionalen Sprache.*
(Computer Specification using a Functional Language).
S. Streitz. Technical report B-48, Bonn, 1985. (In German.)

Strunk **The Elements of Style.**
William Strunk, Jr. and E. B. White. MacMillan, 1979.

11 elementary rules of English usage, 11 elementary rules of composition, 21 suggestions relating to style, and a short section on common misusages, all in 92 tiny pages—peerless.

Sufrin **Z Handbook.**
B. A. Sufrin, editor. Oxford, 1986.

TechComm **Technical Communication.**
Newsletter of the Society for Technical Communication (previously the *STWP Review*, published by the Society of Technical Writers and Publishers.) STC, quarterly.

van Diepen *Listed under* Diepen.

Wichmann **Towards a formal specification of floating point.**
B. A. Wichmann. *The Computer Journal*, 32(5):432-436, Cambridge University Press, October 1989.

Specifies floating point using VDM, then compares it to several previous specifications, including [Barrett].

Wright **Documenting a computer architecture.**
Robert E. Wright. *IBM Journal of Research and Development*, 27(3):257-264, IBM, May 1983.

Explains the requirements of an architecture specification and explains how IBM's architecture-control group produces and updates such specifications. Distinguishes between designing an architecture and documenting it; lists and defines several requirements of an architecture specification (that it be understandable, well written, exact, complete, and unified) and explains how IBM's documentation process is designed to achieve each of these requirements.

This is an extraödinarily good introduction to the problems and techniques of architecture documentation. This paper covers a great deal, and is specific enough to be immediately applied. For example, to unsure understandability, they write with a complexity and vocabulary appropriate to college graduates, provide many cross references, provide a comprehensive index, and use great care in coining descriptive terms. To produce a well-written specification, they use two style guides, re-use

previously produced specification components, and use a single editor and a single architect to produce the final draft. The paper describes further techniques for writing natural-language specifications that are simultaneously exact, complete, and readable.

Zave **Operational specification languages.**
P. Zave. In J. Tartar, program chairman, *ACM '83—Computers: Extending the Human Resource*, proceedings of the 1983 ACM annual conference, pages 214-222, ACM, 1983.

Index