Formal Specification of Software

SEI Curriculum Module SEI-CM-8-1.0

October 1987

Alfs Berztiss
University of Pittsburgh

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

Draft For Public Review

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. A **curriculum module** identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in *designing* a course. A **support materials** package includes materials helpful in *teaching* a course. Other materials under development include model curricula, textbooks, educational software, and a variety of reports and proceedings.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Comments on SEI educational publications, reports concerning their use, and requests for additional information should be addressed to the Director of Education, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Comments on this curriculum module may also be directed to the module author.

Alfs Berztiss
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

Formal Specification of Software

Contents

A support materials package, SEI-SM-8, is available for this module.

Formal Specification of Software

Module Revision History

Version 1.0 (October 1987) Draft for public review

Formal Specification of Software

Capsule Description

This module introduces methods for the formal specification of programs and large software systems, and reviews the domains of application of these methods. Its emphasis is on the functional properties of software. It does not deal with the specification of programming languages, the specification of user-computer interfaces, or the verification of programs. Neither does it attempt to cover the specification of distributed systems.

Philosophy

The term *specification* has various interpretations. Under one interpretation, specification is the process of producing documents that prescribe the requirements, design, behavior and other characteristics of a system or system component [Standard83]. Under this interpretation specification also denotes all the documents produced by the process. A requirements statement defines what a software system is to do, and a design document describes how the system is to do this. However, requirements are generally defined by iterating through two stages. The first stage produces an informal statement in natural language. This statement is then translated, as far as feasible, into a precise language defined by formal syntax and semantics. For lack of a separate term to describe this formal statement we shall call it specification, or, when emphasis is needed, formal specification. In our usage, then, specification denotes (1) the process of producing various characterizations of software, and (2) the product of a specific stage of this process, namely a formal statement of what a software system is to do, where the other products of the specification process are an informal requirements statement and a design. To avoid unnecessary repetition this document uses the term specification to refer to the stage of the specification process that produces specifications.

Central to the development of a large software system is a contract between the developer and a client. This document expresses what the system is to accomplish in precise terms. Parts of the document can be expressed formally, *i.e.*, in a language that has formally defined syntax and semantics. Other parts cannot be so expressed. Our concern here is limited to the formal component.

This module will

- survey a range of formal specification methods;
- establish a taxonomy of software, and relate specification methods to this taxonomy;
- expose students to a broad range of examples of actual formal specifications;
- introduce validation of specifications; and
- relate formal specification to topics such as knowledge representation, verification, transformation of specifications into programs, and reusability.

This module introduces material necessary to understand current trends in the software development process, which are in the direction of increased formalism. It should be a prerequisite of modules that deal with design. There is a close relationship between requirements analysis, formal specification, and design. This suggests that the material dealing with these topics should be studied at about the same time. Indeed, the three topics can be the basis for an introductory course in software engineering.

Objectives

There is an abundance of literature on formal specification, with a variety of different approaches developed by different groups, such as programming methodologists, information system developers, and control system developers. One purpose of this module is to introduce some measure of uniformity.

A student who has mastered the material of this module can be expected to

- understand the central role of formal specification in the software development process;
- be able to partition a system into components and apply appropriate specification methods to these components;
- have participated in a number of specification exercises, at least one of which has been a group project, and to have participated in the validation of specifications.

Because of the centrality of this module, ideally an entire semester should be spent on it. However, it can be combined with requirements analysis and design in a single course. The group project may have to be omitted then, but it must be understood that the experience of working in a team is an important component of education in software engineering, and this experience should still be provided as part of some other module.

Prerequisite Knowledge

The student must have sufficient experience to be able to appreciate the need for proper specifications. This experience may have come from the *ad hoc* development of a software system of some complexity or, better still, from attempts to modify a poorly documented and poorly modularized system. Typically, the student should have written programs containing at least 500 lines of source code, and should have experience with multiple implementations of standard data structures such as stacks and trees.

The student must have an understanding of discrete mathematics at least equivalent to that provided by a three-credit-hour college course. The need here is for mathematical maturity, rather than specific course content. Individual topics, such as predicate logic with quantification, can be introduced as required in the module itself.

There needs to be some understanding of the place of formal specification in the total process of software characterization. [Rombach87] deals with this topic in some detail.

Module Content

Outline

I. Principles of Formal Specification
  1. Definition of formal specification
  2. Requirements and specification
    a. Client-specifier interaction
    b. Abstraction of domain-specific concepts
    c. Modularization
    d. Validation
  3. Specification and design
    a. Declarative and operational styles
    b. Design as algorithm selection
    c. Transformational development of software

II. Formal Specification of Programs
  1. Axiomatic specification
  2. Abstract models
  3. Set theory
  4. Predicate logic
  5. Programming languages in specification
  6. Evaluation of the specification methods
    a. Abstract data types
    b. Data transformers
  7. Examples
    a. Data types: Nat0, set, stack, queue, etc
    b. Data transformer: Text formatter

III. Formal Specification of Persistent Systems
  1. Persistent data bases
    a. Nature of information systems
    b. Conceptual schema
    c. Behavioral aspects
    d. Temporal and spatial aspects of information systems
    e. Specification of knowledge bases
    f. Specification methodologies
  2. Processes
    a. Sequencing of events
    b. Synchronization of processes
    c. Real time
    d. Interaction with sensors
    e. Fail-safe behavior
    f. Specification methodologies
  3. Information-control systems
    a. The SF specification language
    b. Example: Library system
    c. Example: Elevator

IV. The Process of Formal Specification
  1. Mechanics of specification
    a. Specification teams
    b. The specification development log
    c. Graphical aids and other documentation
    d. An infrastructure for specification
  2. Validation and verification
    a. Formal methods
    b. Walkthroughs
    c. Executable specifications
  3. Reusability of specifications

Annotated Outline

The detailed outline of this module is given in a declarative style, *i.e.*, as a description of the various aspects of formal specification of software, rather than as an imperative prescription.

General references: [Berg82], [Birrell85], [Cohen86], [Freeman84], [Gehani86], [Liskov79], [Ramamoorthy78], [Specs87].

I. Principles of Formal Specification

1. Definition of formal specification

Central to the software development process is a contract between a client and a software developer. The contract is prealgorithmic in that it defines the observable properties of a software system without defining the methods that are to provide it with these properties. This document has to capture both functional and non-functional properties of the software system to be developed. Functional properties define the outputs of the system; non-functional properties relate to the processes by which the outputs are obtained. An example of a functional property is the requirement that no output line of a text formatter be longer than 132 characters. The requirement that an elevator system respond to a call

within 10 seconds for 95% of all calls is non-functional. Some parts of the contract can be expressed in a language with formal syntax and semantics—they constitute a formal specification.

In principle, functional properties can be specified formally, but we do not have adequate languages for the formal specification of many of the non-functional components of the contract. Still, as much as possible should be formally specified, for three reasons. First, a contract should be unambiguous, and we need the precision of a formal specification language to ensure this. Second, formalization imposes a uniform style on the contract. The third reason relates to verification, which establishes that a software system is consistent with the contract. Therefore, if the final software product is to be verified, its specification has to be comparable with the product, and, because code is written in formally defined languages, specifications also have to be formal. Note, however, that in order to avoid a bias towards a particular implementation, a specification language should be independent of the language of implementation of the software system.

References: [Anderson84], [Balzer79], [Freeman84], [Heininger80], [Parnas77].

2. Requirements and specification
   a. Client-specifier interaction
   b. Abstraction of domain-specific concepts
   c. Modularization
   d. Validation

Requirements analysis and specification rarely follow each other in strict sequence. Rather, formal specification is an interactive process in which clarifications of the requirements document have to be sought from the client, and the process of clarification may well indicate defects in the requirements definition. A common vocabulary is established in this interaction, and the most important entities of the vocabulary become the data types for the application, *i.e.*, the concrete entities of the application are turned into abstract components of the specification. Abstract data types are often identified with modules; modularization then may take place at some point in the requirements analysis or formal specification stage.

Up to some limit, the greater the investment in the testing of a specification, the lower the total software development cost. Hence, while a specification is being developed, it should be checked against the client's requirements to determine that it reflects them faithfully. This is validation. Validation will suggest changes in the specification; such changes are easier to implement in a modularized specification.

References: [Beierle84], [Birrell85], [Boehm76], [Parnas72].

3. Specification and design
   a. Declarative and operational styles
   b. Design as algorithm selection
   c. Transformational development of software

Specifications follow either a declarative or an operational style. A declarative specification describes the result of an operation with no reference to the operation; an operational specification defines the process by which the result is obtained. Sometimes a declarative definition can be so complicated that the only sensible way to describe the result is to describe the process. However, the description of the process should still be at an abstract level. One advantage of the declarative style is that it gives the designer complete freedom in algorithm selection, but the operational style makes it possible to turn specifications into software by means of transformations, *i.e.*, to bypass the design phase.

References: [Bauer81], [Beierle84], [Finance84], [Gries81], [Liskov86], [Olive86], [Partsch83], [Partsch86], [Zave84].

II. Formal Specification of Programs

1. Axiomatic specification

The algebra of sets can be defined by means of axioms that give precise meaning to operations such as union and complementation. But set algebra need not be regarded as a mathematical system alone. We can interpret sets and operations on sets as a data type, and the axioms then provide a formal specification of this data type. This notion has been extended to what are commonly known as data structures. The method is surveyed in [Berztiss83]; further examples can be found in [Cohen86] and [Manna85]; [Ehrig85] provides a comprehensive treatment. [Burstall81] deals with the combination of axiomatic specification with other specification styles—[Goguen86] is a technical overview of later development of this approach.

2. Abstract models

Implementation of data types specified by axioms is rather difficult. Also, the axiomatic specification of some data structures requires an inordinately large number of axioms. Abstract modeling gets around these difficulties by the selection of an abstract implementation of a data type in terms of a more basic type, *e.g.*, a stack in terms of a sequence, and a stack operation is then described by the effects it has on the sequence. This approach is contrasted to axiomatic specification in [Liskov79]. It is used in the Alphard [Shaw81] and CLU [Liskov86] languages, which permit both specification (by abstract modeling) and implementation of data types. A dif-

ferent approach is to associate a state with each data object and define operations in terms of state changes [Claybrook82]. The Vienna Development Method (VDM) is a very elaborate language based on the abstract model approach. An introduction with good examples can be found in [Cohen86]; some very large application studies are described in [Bjorner82]; [Jones78] is the reference manual.

3. Set theory

A group at Oxford University has developed the specification language Z, which is based on set theory and logic. A rather complicated symbolism is built up by defining relations, functions, orders, sequences, and bags in terms of sets. This approach can be regarded as a variant of the abstract model approach, with set theory providing the abstract model. [Abrial80] is an early exposition of Z; [Hayes87] contains extensive examples.

4. Predicate logic

A computational unit (function, procedure) can be specified by predicates describing the output from the unit. Generally two predicates are needed—one defines the appearance of the output, the other relates the output to the input. For example, sorting is specified by predicates that indicate that the output of the unit is in fact sorted and that it is a permutation of the input. An excellent introduction to this approach, and how it relates to program verification, can be found in [Gries81]. Predicate logic is combined with axiomatic specification in Larch. This specification language has a common shared language in which data types, called traits in Larch, are defined axiomatically, and interface languages (one for each implementation language under consideration), in which operations from the traits are used in predicative specification of program units. For introductions to Larch see [Horning85], [Guttag85], and [Liskov86]; more detail can be found in [Guttag86a] and [Guttag86b].

5. Programming languages in specification

We have already mentioned Alphard and CLU, which can be regarded as languages for both specification and programming. Use of Ada for specification has been advocated [Ada85], but this has to be approached with some caution. If the specification is syntactically correct, then we already have a program. Now, under the assumption that specification is prealgorithmic, a program should not be regarded as a specification, but programs, too, leave low-level algorithms undefined (*e.g.*, the algorithm for multiplication implied by the expression *a*b*). Indeed, since both formal specifications and programs are written in languages with formal syntax and semantics, the distinction between the two can become blurred. The logic programming language Prolog is another candidate for consideration as a specification language [Kowalski85]. For an interesting example of a Prolog specification see [Veloso85]. Functional programming has also been used in specification [Turner85, Henderson86]. Both logical programs and functional programs are regarded as their own specifications, with "implementation" of a functional specification considered merely as a search for greater efficiency.

6. Evaluation of the specification methods

a. Abstract data types

b. Data transformers

Let us partition data types into three kinds. First are some fundamental data types, such as booleans, reals, and integers. Next are basic structuring types, namely sets, bags, sequences, and maps (finite functions). The third kind consists of devices; they are used for the implementation of algorithms in the design phase. For the fundamental data types and the basic structuring types axiomatic specification, abstract models, and the set theoretical approach are more or less equally easy to apply. The choice is then determined by other considerations. Either the axiomatic or the set theoretical approach should be used when the program derived from the specification is to be formally verified; the abstract model permits easier implementation. As regards devices, some can be defined as special cases of sequences, *e.g.*, stacks and queues, and arrays (sequences of sequences). Other devices need to be defined independently, *e.g.*, binary trees. With some devices the number of axioms can become too large for axiomatic specification to be practicable, but the abstract model approach and Z are both still applicable.

Only some of the operations of an abstract data type are regarded as basic, *e.g.*, an operation that reads the top element of a stack. An operation that, say, replaces the top two elements of a stack with a single new element is a derived operation. However, the distinction between basic and derived operations is not sharp. In one context we might regard matrix multiplication basic, in another derived. Even the classification of types is not sharp: text may be regarded as fundamental or as a special case of the sequence.

Still, we shall make a distinction between basic and derived operations of a data type, and refer to the latter as data transformers. A data transformer accepts one or more objects as input and transforms them into output objects. The specification has to describe the output and relate the output to the input. For this predicate logic is the best choice, possibly applied in the context of Larch or Z.

Another way of looking at data transformation is in terms of data streams. Then the input to a data transformer is one or more input streams. The out-

put may be a single value (the inner product of two vectors), a new data stream (formatted text), or several streams (a phase of a merge sort). The data streams—the preorder sequence of a binary tree, sorted values from a heap, a pseudorandom number sequence—can be produced by generators. A discussion of generators can be found in [Griswold81].

A data transformer may be specified in a number of different ways. Consider a spelling checker that takes a text (intext) and a dictionary, and generates the set of words found in the text but not in the dictionary. At one level, the specification may simply be a predicate that defines the output as the set difference of the input text and the dictionary, regarded as sets of words. At another level, the spelling checker may be regarded as being composed of generators that produce, in turn, text separated into words (split), a sorted word list (sort), the same list with duplicates removed (reduce), and the list with words found in the dictionary removed (diff). Each generator can be defined by predicates, and the spelling checker as a whole can be defined by functional composition: errors = diff (reduce (sort (split (intext))), dictionary). Under the first approach it has to be verified that an implementation is in fact consistent with the specification. The functional composition of the second approach is its own specification.

Unfortunately it is an overspecification in that the particular sequencing of the operations is really an algorithm. However, specifications written in terms of predicates tend to be very complex, and the better readability of a specification as a composition of operations may outweigh the loss of freedom brought about by the overspecification. It has to be emphasized that the writing of the predicates that specify a text formatter, say, is a very difficult task, and the result is very difficult to read. Indeed, the reading difficulty may prevent adequate validation of a specification. In such a case it may be worthwhile to sacrifice freedom of choice for better readability [Naur82].

7. Examples
   a. Data types: Nat0, set, stack, queue, etc
   b. Data transformer: Text formatter

III. Formal Specification of Persistent Systems

1. Persistent data bases
   a. Nature of information systems
   b. Conceptual schema
   c. Behavioral aspects
   d. Temporal and spatial aspects of information systems
   e. Specification of knowledge bases
   f. Specification methodologies

Section II relates to the generation of results by intensive computation. Such computations are the concern of classical programming. A persistent data base, on the other hand, is a resource that is updated and consulted over an extended period of time. A system based on a persistent database, with capabilities for changing the database and responding to queries, is known as an information system. For discussions of the issues involved in the specification of information systems see [Bubenko80, Brodie84, Borgida85a, Jardine84]. We shall survey the issues briefly here.

A conceptual schema describes the organization of the database. The schema may consist of relation tables, or a structure of entity sets and relationships [Chen76], or a collection of data types that consist of sets and functions [Berztiss86a, Berztiss86b]. Events cause changes in the database. A behavioral model superimposed on the conceptual schema relates to the database changes. It defines either the valid states of the database (declarative approach) or the valid events (operational approach) [Olive86].

An example of a declaration is the formal expression corresponding to the statement "the salary of an employee may never be greater than the salary of the supervisor of the employee". This approach has the advantages that deductive processes may be applied to the declarations, and therefore it is sometimes called the deductive approach. Moreover, the methods by which database integrity is maintained are left open. Under the operational approach one would check that a proposed new salary for an employee is no larger than that of the supervisor and no smaller than those of the subordinates of this employee, and the new salary would only take effect if these preconditions were satisfied. An operational specification is easier to convert into a prototype implementation. An operational specification may also be easier to formulate, but not always.

Suppose that the data base consists of finite sets and functions. A distinction needs to be made between data types of the application, such as *employee*, *library catalog*, and the data types *set* and *function*, which implement the data types of the application. The operational specification indicates under what conditions the sets and functions may change, but the actual changes are brought about by set and function operations. The latter would have been specified by the techniques of Section II.

An information system should be able to deal with time references in the database and with sequencing of events (temporal aspect), and with distribution of the database over different sites, particularly in an office setting (spatial aspect), but methodologies for specifying temporal and spatial aspects are still under development. For some approaches to the specification of office automation see [Gibbs83,

Chang85]. An information system that can cope with incomplete and inconsistent data, and includes inference-making capabilities, is known as a knowledge system. Specification methodologies for knowledge systems are in a research stage, but progress is being made toward their development [Borgida85b, Brodie86].

The entire topic of specification of information systems is still unsettled. Numerous specification methodologies have been proposed, with varying degrees of formality [Olle82, Olle83, Olle86, Brodie84, Furtado86]; we have tried to distill features from all these methodologies into a composite approach called SF—it is described in Section III.3. Note that Larch has been used to specify an information system [Wing87], but its general suitability for this purpose remains to be established.

2. Processes
   a. Sequencing of events
   b. Synchronization of processes
   c. Real time
   d. Interaction with sensors
   e. Fail-safe behavior
   f. Specification methodologies

A process is the controlled evolution of a system in time, where the controlling actions normally depend on the states of a database. The controlled system is generally external to the controlling software, *e.g.*, an elevator. We shall refer to a system comprising a database and a process or a system of processes as an information-control system. Again it is useful to think in terms of events, where the events change the database or the controlled system. The simplest control operation is the ordering of events in sequences. The sequence of operations of a data transformer may be indicated by functional composition, but the sequencing of events in an information-control system needs to be indicated by more complicated control mechanisms such as path expressions or traces (declarative approach) [Campbell74, Furtado85], or message passing (operational approach) [Berzins85].

A sequence of events defines a process, and the specification of a system may have to define how processes are synchronized. The control mechanisms mentioned above are adequate for this purpose as well. Sometimes, however, events must take place at a specific time (in the United States, clocks were advanced one hour at midnight on April 4, 1987) or after a specific delay (holders of overdue library books must be reminded after a grace period expires). The specification methodology must be able to deal with such real-time aspects. Further, a controller receives inputs, by means of sensors, from the system it controls. The specification has to indicate what happens when a sensor fails. The specification for the controlling software does not have to define how the controlled system behaves when the controller fails, but it does have to define how the controller is to start up again after a failure. Issues relating to fault-tolerance are discussed in [Avizienis85, Leveson87].

Most of the aspects discussed here have related to performance rather than functionality. We should therefore determine which non-functional requirements can and which cannot be specified formally. A general characterization is a major research task, but the distinction can be made in individual instances. Consider a bank of elevators, a module for the operation of an individual elevator, and a dispatcher module for the control of the entire bank of elevators. The operation of a single elevator is determined by its users and the dispatcher, and a specification can be written quite easily. However, the dispatcher is to see to it that, for example, an elevator will reach a caller within 10 seconds for 95% of all calls. At this time the only nontrivial formalization of this requirement seems to be algorithmic, but an algorithm is a component of design rather than specification.

The three most prominent examples of specification languages for systems that include control features are Gist [Feather87], PAISLey [Zave86], and MSG.84 [Berzins85, Berzins86], but see also [Dasarathy85]. These approaches emphasize the executability of specifications. Also, synchronization of processes was not the primary consideration in the design of these specification languages. A specification method that does emphasize synchronization is Hoare's CSP [Hoare78]; it has been applied to the elevator problem [Schwartz87]. Another approach has been to use temporal logic, which is an extension of classical logic that enables it to deal with time. Here the primary concern is the suitability of the specification methodology for the verification of an implementation of a system of processes —[Manna81] is a good exposition. A common feature of all these approaches is the minor attention given to the specification of database operations. Specification of distributed systems is a specialized topic that is not included in this module—for a survey see [Alford85], but note that Z may be used to specify distributed systems [Hayes87].

3. Information-control systems
   a. The SF specification language
   b. Example: Library system
   c. Example: Elevator

It was noted earlier that there is a large variety of methodologies for the specification of information systems, but rarely do they address control issues. On the other hand, languages for the specification of

control processes do not deal adequately with database issues. The specification language SF (Set-Function) [Berztiss86a, Berztiss86b] has features for dealing with both information and control; it is a language for the specification of information-control systems. It does not attempt to deal with the specification of basic data types—when a queue is needed, it is imported, under the assumption that it has been specified in some other framework, say Larch.

Specifications in SF of a library system and of an elevator are to be found in the support material package for this module.

IV. The Process of Formal Specification

1. Mechanics of specification

a. Specification teams

b. The specification development log

c. Graphical aids and other documentation

d. An infrastructure for specification

Specifications of large systems are themselves large and complex, and their development is necessarily a group activity. In fact, writing the specification may take longer than building the system from that specification. Further, requirements typically change while a specification is being produced. It is essential that all assignments to members of a specification team and all requirements changes are documented in a log kept by the specification team. A register of other documentation should also be maintained. Such documentation may contain functionality diagrams of basic data types and devices, data flow diagrams, E-R (Entity-Relationship) diagrams, state transition diagrams, Petri nets, etc. Specification teams should be provided with a proper infrastructure to perform their task efficiently, *i.e.*, they should have access to electronic conferencing facilities and the like, and they should be provided with tools for configuration control, cross-referencing, etc.

References: [Bidoit86], [Chen76], [Martin85], [Peterson81], [Ramamoorthy86], [Reisig85].

2. Validation and verification

a. Formal methods

b. Walkthroughs

c. Executable specifications

The consistency and sufficient completeness of algebraic specifications can be established, but we regard this as part of verification, and verification is not emphasized in this module. Instead, validation is stressed, particularly the static analysis technique known as walkthroughs. Moreover, prototype implementation of the specifications of information-control systems is rather easy, and dynamic test methods can be applied to the prototype. If the prototype is derived from the specification by "correctness-preserving" transformations, then it must necessarily be consistent with the specification, *i.e.*, there is no need for further verification.

References: [Balzer85], [Berg82], [Bouge85], [Henderson86], [Kemmerer85], [Turner85], [Yourdon86].

3. Reusability of specifications

Recent work on reusability, particularly by Biggerstaff and Richter, suggests that reuse of specifications is more practicable than reuse of designs or code. A possible application of reusable specifications is in the software factory. So far, reusability in this context has been confined to code, but it should be extended to specifications. The reusability of specifications depends on modularization. For example, when a text formatter is being specified, a specification module for the text data type should already exist. This module is retrieved, and operations from it used in the specification of operations to define the application. Any of the latter that are deemed to be of general interest can be added to the text module. Moreover, it should be indicated in the documentation of the text module that it has been used in the specification of the text formatter. A type of semantic net is thus created that should help in retrieving components for reuse at a later time. Even research on the reuse of specifications has not properly begun yet.

References: [Biggerstaff87], [Matsumoto84].

Teaching Considerations

Suggested Schedules

As already indicated in the statement of objectives, an entire semester, nominally 40 hours, should ideally be spent on this module. The following rough breakdown is suggested:

- Principles of Formal Specification: 10 hours
- Formal Specification of Programs: 10 hours
- Formal Specification of Persistent Systems: 10 hours
- The Process of Formal Specification: 10 hours

No finer breakdown will be attempted. Too much depends on the background of the instructor, the composition of the class, and the rapport between instructor and class. Moreover, if a major specification project is undertaken, topics in different sections would have to be intermixed, so that lecture material could support project development effectively. The material on Principles of Specification is to come in large part from [Rombach87].

If teaching of specification is combined with requirements analysis and design, a fair amount of integration can be undertaken, particularly of requirements analysis and specification, and the breakdown below is suggested:

- Principles of Software Development: 10 hours
- Specification and Design Methodologies: 20 hours
- The Process of Software Development: 10 hours

Support Materials

The support materials package for this module contains specifications of the library system and elevator examples written in the SF specification language for information-control systems, as well as a specification of the text formatter expressed in terms of predicates. Specifications of these systems produced by other approaches can be found in [Specs87]. With regard to the elevator example, the finite-state transition diagram and Petri net that were actually used to come to an understanding of the problem (and not added as a decoration after all work had been done) are also included.

Teams of students in a software specification course at the University of Pittsburgh specified a student registration system or an elevator controller. The teams kept logs of their activities; an example is included.

Projects and Exercises

Several projects and exercises are suggested in the support materials package for this module. Some of the projects are:

- a student registration system
- a system of coin-operated luggage lockers at an airport or railroad station
- a dry-cleaning/formalwear-rental business

Smaller specifications can be designed for:

- a car cruise-control system;
- a telephone dialing system
- a traffic light system
- the n-queens problem

It is highly recommended that the instructor do several specification exercises before teaching this material.

Bibliography

Although there are several good books that deal with the specification of what we have called basic operations of abstract data types and data transformers, there is yet no comprehensive text that would cover the specification of information-control systems as well. One of the purposes of this module is to integrate the separate developments; a textbook to support this endeavor should become available in 1988. For the time being, however, instructors of this module will have to read rather extensively. Some of the papers have been marked essential reading. These papers should be read by the instructor, who can then decide what should be assigned to the class to read on the basis of the background of the students.

Abrial80

Abrial, J. R., S. A. Schuman, and B. Meyer. "Specification Language." In *On the Construction of Programs*, R. M. McKeag and A. M. Macnaghten, eds. Cambridge, England: Cambridge University Press, 1980, 343-410.

This is an early description of Z, a specification language based on sets.

Ada85

Goldsack, S. J., ed. *Ada for Specification: Possibilities and Limitations*. Cambridge, England: Cambridge University Press, 1985.

Although primarily an investigation of Ada as a specification language, this book is also a valuable survey of specification languages in general, and of the process that transforms a specification into code. The view is taken that Ada can be used for both coding and specification, but that two different sets of semantics then have to exist for the two contexts. At least some parts of this book should be read.

Alford85

Alford, M. W., J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider. *Distributed Systems: Methods and Tools for Specification*. Berlin: Springer-Verlag, 1985. Springer-Verlag Lecture Notes in Computer Science, No. 190.

Notes for a course on the specification of distributed systems, *i.e.*, a subset of what we have called information-control systems in this module.

Anderson84

T. Anderson, ed. *Software Requirements, Specification and Testing*. Oxford, England: Blackwell, 1984.

A collection of non-technical papers on the topics listed in the title.

Avizienis85

Avizienis, A. "The N-version Approach to Fault-Tolerant Software." *IEEE Trans. Software Eng. SE-11* (1985), 1491-1501.

Abstract: *Evaluation of the N-version software approach to the tolerance of design faults is reviewed. Principal requirements for the implementation of N-version software are summarized and the DEDIX distributed supervisor and testbed for the execution of N-version software is described. Goals of current research are presented and some potential benefits of the N-version approach are identified.*

Contains strong arguments that link fault-tolerance to effective specification. Should be examined.

Balzer79

Balzer, R., and N. Goodman. "Principles of Good Software Specification and their Implication for Specification Languages." *Proc. IEEE Conf. Specifications of Reliable Software.* Silver Spring, Md.: IEEE Computer Society Press, 1979, 58-67. Reprinted in [Gehani86], 25-39.

Abstract: *Careful consideration of the primary uses of software specifications leads directly to three criteria for judging specifications, which are then used to develop eight design principles for "good" specifications. These principles, in turn, imply a number of requirements for specification languages that strongly constrain the set of adequate specification languages and identify the need for several novel capabilities such as historical and future references, elimination of variables, and result specification.*

A catalog of eight principles and eighteen implications for the design of specification languages. Essential reading.

Balzer85

Balzer, R. "A 15 Year Perspective on Automatic Programming." *IEEE Trans. Software Eng. SE-11* (1985), 1257-1268.

Abstract: *Automatic programming consists not*

only of an automatic compiler, but also some means of acquiring the high-level specification to be compiled, some means of determining that it is the intended specification, and some (interactive) means of translating this high-level specification into a lower-level one which can be automatically compiled.

We have been working on this extended automatic programming problem for nearly 15 years, and this paper presents our perspective and approach to this problem and justifies it in terms of our successes and failures. Much of our recent work centers on an operational testbed incorporating usable aspects of this technology. This testbed is being used as a prototyping vehicle for our own research and will soon be released to the research community as a framework for development and evolution of Common Lisp systems.

Balzer summarizes his experiences and observations regarding the transformational development of software. Should be examined.

Bauer81

Bauer, F. L., *et al*. "Programming in a Wide Spectrum Language: A Collection of Examples." *Science of Comp. Programming 1* (1981), 73-114.

Abstract: *The paper exemplifies programming in a wide spectrum language by presenting styles which range from non-operative specifications—using abstract types and tools from predicate logic as well as set theory—over recursive functions, to procedural programs with variables. Besides a number of basic types, we develop an interpreter for parts of the language itself, an algorithm for applying transformation rules to program representations, a text editor, and a simulation of Backus' functional programming language.*

An introduction to the CIP (Computer-aided Intuition-guided Programming) approach. The same language is used for specifications and machine-oriented programs, with transformations converting the former into the latter.

Beierle84

Beierle, C., M. Gerlach, R. Gobel, W. Olthoff, R. Raulefs, and A. Voss. "Integrated Program Development and Verification." In *Software Validation*, H. L. Hausen, ed. Amsterdam: North-Holland, 1984, 189-205.

Abstract: *A survey of an integrated program development and verification support environment is prtesented. The system supports the entire range from requirements definitions to verified programs.*

A sophisticated set of tools is proposed that supports the entire system development process from requirements definition to verified implementation. (By now the tools have been implemented. The result is an impressive system that has not received the recognition it deserves.)

Berg82

Berg, H. K., W. E. Boebert, W. R. Franta, and T. G. Moher. *Formal Methods of Program Verification and Specification*. Englewood Cliffs, N. J.: Prentice-Hall, 1982.

Here the primary concern is verification, with specification considered only as it relates to verification. Still, the chapter on specification gives a useful overview of specification of the objects of classical programming, and should be read. The bibliography is extensive (166 entries).

Berzins85

Berzins, V., and M. Gray. "Analysis and Design in MSG.84: Formalizing Functional Specifications." *IEEE Trans. Software Eng. SE-11* (1985), 657-670.

Abstract: *Model building is identified as the most important part of the analysis and design process for software systems. A set of primitives to support this process is presented, along with a formal language, MSG.84, for recording the results of analysis and design. The semantics of the notation is defined in terms of the actor formalism, which is based on a message passing paradigm. The automatic derivation of a graphical form of the specification for user review is discussed. Potentials for computer-aided design based on MSG.84 are indicated.*

Defines MSG.84, which supports the specification of control systems via the actor formalism of message passing.

Berzins86

Berzins, V., M. Gray, and D. Naumann. "Abstraction-Based Software Development." *Comm. ACM 29* (1986), 402-415.

Abstract: *A five-year experience with abstraction-based software-development techniques in the university environment indicates that the investment required to support the paradigm in practice is returned in terms of greater ability to control complexity in large projects—provided there exists a set of software tools sufficient to support the approach.*

Reports on classroom experience with MSG.84, the specification language introduced in [Berzins85].

Berztiss83

Berztiss, A. T., and S. Thatte. "Specification and Implementation of Abstract Data Types." In *Advances*

in Computers, Vol. 22. New York: Academic Press, 1983, 295-353.

A survey of the equational algebraic specification of abstract data types.

Berztiss86a

Berztiss, A. "The Set-Function Approach to Conceptual Modeling." In *Information System Design Methodologies: Improving the Practice*, T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, eds. Amsterdam: North-Holland, 1986, 107-144.

Abstract: *We examine the design of information systems and develop a methodology for the construction of conceptual schemas. We call it the set-function (SF) methodology, and use it to express a conceptual schema for the IFIP Working Conference example. Novel features of our approach: (i) a deliberate attempt to integrate the specification of information systems with the formalisms of data type specification in programming methodology, (ii) simultaneous concerns with theoretical foundations and simplicity of use, (iii) separation of static and dynamic aspects of the specification to allow one to use only as much of the methodology as is necessary for a particular task, (iv) a strong concern with reliability of the conceptual schema being defined, including considerations of rapid prototyping, and (v) a design that should foster the use of fourth and fifth generation system development tools.*

Introduces the SF specification language and uses it on the IFIP Working Conference case study. The specification has a major flaw in that it is not modularized. The paper contains a fairly extensive discussion of properties of specifications of information systems in general.

Berztiss86b

Berztiss, A. "Data Abstraction in the Specification of Information Systems." *Proc. IFIP World Congress 1986.* Amsterdam: North-Holland, 1986, 83-90.

Abstract: *Four classes of computational activities are identified, namely changes in an information base, look-ups, computation of function values by computational rules, and processes, and it is argued that they require different specification mechanisms. Our particular concern is information bases, which we define as collections of sets and functions. Their changes are specified in terms of events, and temporal aspects are taken care of by a fully separate "responder". The methodology is illustrated by the specification of a system for managing checking accounts by a bank.*

The SF language is used to specify an account handling system for a bank. Some essential differences between devices, such as queues and binary trees, and the data types of an information system are pointed out. Should be read because it is a fairly coherent statement of the philosophical foundations of this module.

Bidoit86

Bidoit, M., C. Choppy, and F. Voisin. "The ASSPEGIQUE Specification Environment." In *Recent Trends in Data Type Specification*, H. J. Kreowski, ed. Berlin: Springer-Verlag, 1986, 54-72. Springer-Verlag Informatikfachberichte, No. 116.

Describes an environment for the development of algebraic specifications. In particular, the environment contains a graphical tool that displays the functionality diagram for an abstract data type.

Biggerstaff87

Biggerstaff, T., and C. Richter. "Reusability Framework, Assessment, and Directions." *IEEE Software 4*, 2 (March 1987), 41-48.

Abstract: *Reusability is widely believed to be a key to improving software development productivity and quality. The reuse of software components amplifies the software developer's capabilities. It results in fewer total symbols in a system's development and in less time spent on organizing those symbols.*

However, while reusability is a strategy of great promise, it is one whose promise has been largely unfulfilled.

Expresses the belief that the greatest payoffs can be expected from reuse of specifications by means of what the authors call semantic binding. Should be read.

Birrell85

Birrell, N. D., and M. A. Ould. *A Practical Handbook for Software Development*. Cambridge, England: Cambridge University Press, 1985.

A rich source of examples of the use of various communication tools, *e.g.*, diagrammatic representations such as data flow diagrams and Petri nets.

Bjorner82

Bjorner, D., and C. B. Jones. *Formal Specification and Software Development*. Englewood Cliffs, N. J.: Prentice-Hall, 1982.

An introduction to the Vienna Development Method (VDM), together with a rich selection of examples of its application. VDM has been used with considerable success in the specification of data transformers.

Boehm76

Boehm, B. "Software Engineering." *IEEE Trans. Computers C-25* (1976), 1226-1241.

Abstract: *This paper provides a definition of the term "software engineering" and a survey of the current state of the art and likely future trends in the field. The survey covers the technology available in the various phases of the software life cycle —requirements engineering, design, coding, test, and maintenance—and in the overall area of software management and integrated technology-management approaches. It is oriented primarily toward discussing the domain of applicability of techniques (where and when they work), rather than how they work in detail. To cover the latter, an extensive set of 104 references is provided.*

Provides data which show that error correction is least costly when the errors are detected and repaired during requirements analysis and specification.

Borgida85a

Borgida, A. "Features of Languages for the Development of Information Systems at the Conceptual Level." *IEEE Software 2*, 1 (Jan. 1985), 63-72.

Abstract: *Conceptual modeling languages make information systems easier to design and maintain by using vocabularies that relate naturally and directly to the "real world" of many computer applications.*

A useful set of principles for the design of specification languages for information systems. The author emphasizes the important property that specifications of information systems can be automatically translated into implementations. Should be read.

Borgida85b

Borgida, A., S. Greenspan, and J. Mylopoulos. "Knowledge Representation as the Basis for Requirements Specification." *Computer 18*, 4 (April 1985), 82-91.

Abstract: *Specification of many kinds of knowledge about the world is essential to requirements engineering. Research on knowledge representation in artificial intelligence provides a wealth of relevant techniques that can be incorporated into specification languages.*

An investigation of the relationship between the knowledge representation techniques of artificial intelligence and the specification of information systems. Required reading for one view of the direction in which the methodology for the development of information systems should be moving.

Bouge85

Bouge, L., N. Choquet, L. Fribourg, and M. C. Gaudel. "Application of Prolog to Test Sets Generation from Algebraic Specifications." In *Proc. TAPSOFT '85, Vol. 2*. Berlin: Springer-Verlag, 1985, 261-275. Springer-Verlag Lecture Notes in Computer Science, No. 186.

Abstract: *We present a method and a tool for generating test sets from algebraic data type specifications. We give formal definitions of the basic concepts required in our approach of functional testing. Then we discuss the problem of testing algebraic data types implementations. This allows the introduction of additional hypotheses and thus the description of an effective method for generating test sets. The method can be improved by using PROLOG. Indeed, it turns out that PROLOG is a very well suited tool for generating test sets in this context. Applicability of the method is discussed and a complete example is given.*

A specialized paper, but indicative of an interesting mix of theory and practice that characterizes much of the current research on specification.

Brodie84

Brodie, M. L., J. Mylopoulos, and J. W. Schmidt, eds. *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York: Springer-Verlag, 1984.

A collection of papers dealing with the application of information systems to knowledge representation, etc. Should be examined.

Brodie86

Brodie, M. L., and J. Mylopoulos, eds. *On Knowledge Base Management Systems*. New York: Springer-Verlag, 1986.

A collection of papers dealing with the transformation of information systems into knowledge systems. Should be examined, but not as relevant to the specification of information systems as [Brodie84].

Bubenko80

Bubenko, J. "Information Modeling in the Context of System Development." *Proc. IFIP World Congress 1980.* Amsterdam: North-Holland, 1980, 395-411.

Abstract: *The concepts of an information system and information requirements are examined. An appraisal of significant results in the areas of information system specification and of data modeling is presented. A framework for specification of goal-oriented information requirements for an informa-*

tion system is outlined. It is argued that a total requirement specification must include an abstract model of the enterprise. The model should view the application in an extended time perspective. The main part of this paper is concerned with concepts useful for specification of such a model.

This is one of the most widely cited papers on the development of information systems. It is particularly useful for its discussion of time in the specification of information systems. The philosophy underlying the SF specification language of [Berztiss86a] has much in common with the views expressed in this paper. Required reading.

Burstall81

Burstall, R. M., and J. A. Goguen. "An Informal Introduction to Specifications Using CLEAR." In *The Correctness Problem in Computer Science*, Boyer, R. S., and J. S. Moore, eds. Academic Press, London, 1981, 185-213. Also in [Gehani86], 363-389.

Abstract: *Precise and intelligible specifications are a prerequisite for any systematic development of programs, as well as being the starting point for correctness proofs. This paper describes "Clear", a language for giving modular and well-structured specifications; the informal presentation gives examples and sketches the algebraic background.*

This very important paper contains an informal introduction to institutions, which permit different styles to be combined in one specification. Required reading.

Campbell74

Campbell, R. H., and A. N. Habermann. "The Specification of Process Synchronization by Path Expressions." In *Proc. Internat. Symp. Operating Systems, Rocquencourt, 1974*, Gelenbe, E., and C. Kaiser, eds. Berlin: Springer-Verlag, 1974, 89-102. Springer-Verlag Lecture Notes in Computer Science, No. 16.

Abstract: *A new method of expressing synchronization is presented and the motivations and considerations which led to this method are explained. Synchronization rules, given by 'path expressions', are incorporated into the type definitions which are used to introduce data objects shared by several synchronous processes. It is shown that the method's ability to express synchronization rules is equivalent to that of P and V operations, and a means of automatically translating path expressions to existing primitive synchronization operations is given.*

Path expressions have been used to analyze permissible patterns of interleaving of processes in a system. They could be used equally well to analyze sequences of events.

Chang85

Chang, S. K., and W. L. Chan. "Transformation and Verification of Office Procedures." *IEEE Trans. Software Eng. SE-11* (1985), 724-734.

Abstract: *An office procedure is a structured set of office activities for accomplishing a specific office task. A unified model, called office procedure model (OPM), is presented to model office procedures. The OPM describes the relationships among messages, databases, alerters, and activities. The OPM can be used to coordinate and integrate the activities of an office procedure. The OPM also allows the specification of office protocols in an office information system. A methodology for the verification of office procedures is presented. With this methodology, potential problems in office procedure specification, such as deadlock, unspecified message reception, etc., can be analyzed effectively.*

The office procedure model presented in this paper allows the specification of office protocols in an office information system.

Chen76

Chen, P. P. "The Entity-Relationship Model: Toward a Unified View of Data." *ACM Trans. Database Syst. 1* (1976), 9-36.

Abstract: *A data model, called the entity-relationship model, is proposed. This model incorporates some of the important semantic information about the real world. A special diagrammatic technique is introduced as a tool for database design. An example of database design and description using the model and the diagrammatic technique is given. Some implications for data integrity, information retrieval, and data manipulation are discussed.*

The entity-relationship model can be used as a basis for unification of different views of data: the network model, the relational model, and the entity set model. Semantic ambiguities in these models are analyzed. Possible ways to derive their views of data from the entity-relationship model are presented.

This paper introduces the E-R model, which has probably had the most significant influence on subsequent methodologies for the development of information systems. Required reading.

Claybrook82

Claybrook, B. G. "A Specification Method for Specifying Data and Procedural Abstractions." *IEEE Trans. Software Eng. SE-8* (1982), 449-459.

Abstract: *A specification method designed primarily for specifying data abstractions, but suitable for describing procedural abstractions as well, is de-*

scribed. The specification method is based on the abstract model approach to specifying abstractions. Several data abstractions and procedural abstractions are specified and a proof of implementation correctness is given for one of the data abstractions —a symbol table.

The abstract model approach is used to specify the data type of symbol table, and the implementation based on this specification is proven correct.

Cohen86

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

A fairly short (143 pages) introduction that explores some aspects of the electronic office by means of equational algebraic specification and the Vienna Development Method (VDM). Specification of concurrent systems is briefly touched on as well. Listings of centers of current research activity in the more theoretical approaches to specification, and of experimental specification languages are particularly valuable. Essential reading.

Dasarathy85

Dasarathy, B. "Timing Constraints on Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them." *IEEE Trans. Software Eng. SE-11* (1985), 80-86.

Abstract: *This paper examines timing constraints as features of real-time systems. It investigates the various constructs required in requirements languages to express timing constraints and considers how automatic test systems can validate systems that include timing constraints. Specifically, features needed in test languages to validate timing constraints are discussed. One of the distinguishing aspects of three tools developed at GTE Laboratories for real-time systems specification and testing is in their extensive ability to handle timing constraints. Thus, the paper highlights the timing constraint features of these tools.*

Considers timing constraints in telephone dialing.

Ehrig85

Ehrig, H., and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Berlin: Springer-Verlag, 1985.

A comprehensive study of equational algebraic specification of data types. A book for the specialist, but has a very useful bibliography.

Feather87

Feather, M. S. "Language Support for the Specification and Development of Composite Syatems." *ACM Trans. Prog. Lang. and Syst. 9* (1987), 198-234.

Abstract: *When a complex system is to be realized as a combination of interacting components, development of those components should commence from a specification of the behavior required of the composite system. A separate specification should be used to describe the decomposition of that system into components. The first phase of implementation from a specification in this style is the derivation of the individual component behaviors implied by these specifications.*

The virtues of this approach to specification are expounded, and specification language features that are supportive of it are presented. It is shown how these are incorporated in the specification language Gist, which our group has developed. These issues are illustrated in a development of a controller for elevators serving passengers in a multistory building.

This is the most recent in a series of papers dealing with the specification language Gist. Gist is based on 'histories', which correspond to traces of event occurrences.

Finance84

Finance, J. P., M. Grandbastien, N. Levy, A. Quere, and J. Souquieres. "SPES: A Specification and Transformation System." *Proc. 2nd AFCET Software Eng. Conf.* Oxford, England: North Oxford Academic, 1984, 145-151.

Abstract: *The aim of the SPES project is to construct and transform formal specifications of computer problems, in a methodical way. The structure used expresses relations between data and results, within modules called texts. The structure is described using abstract types. Transformations applied to the specification make it possible top modify it, with a view to building a program. This approach, which applies a user assistance system, is illustrated with a simple example.*

Describes a system to assist the software developer in the construction of a specification and in the transformation of the specification into a program.

Freeman84

Freeman, F., and Wasserman, A. I., eds. *Tutorial on Software Design Techniques, 4th ed.*. Silver Spring, Md.: IEEE Computer Society Press, 1984.

A collection of papers that relate primarily to software design, but a number of significant contributions to specification methodology are also included. Good browsing.

Furtado85

Furtado, A. L., and T. S. E. Maibaum. "An Informal Approach to Formal (Algebraic) Specifica-tions." *Computer J. 28* (1985), 59-67.

Abstract: *Formal techniques exist for the crucial specification phase in the design of systems, including database applications. We briefly indicate the potential benefits of the so-called abstract data type discipline and show how it might be made more palatable to the non-mathematician. This is done through the mechanism of traces. This tool is used both as a mechanism for modelling (in an executable manner) the application and as a basis for a methodology which can be used in the development of a formal algebraic specification.*

A very readable introduction to the use of traces of operations as a basis for reasoning about information-control systems.

Furtado86

Furtado, A. L., and E. J. Neuhold. *Formal Techniques for Data Base Design*. Berlin: Springer-Verlag, 1986.

A monograph that exposes designers of information system specifications to new developments in data abstraction and data modeling. One of the few publications in this area that considers control as well. Should at least be examined.

Gehani86

Gehani, N., and A. D. McGettrick, eds. *Software Specification Techniques*. Wokingham, England: Addison-Wesley, 1986.

A collection of 21 papers, most of which have contributed significantly to shaping the field of software specification. Some are listed individually in this bibliography, but the others should also be examined.

Gibbs83

Gibbs, S., and D. Tsichritzis. "A Data Modeling Approach for Office Information Systems." *ACM Trans. Office Info. Syst. 1* (1983), 299-319.

Abstract: *A data model for representing the structure and semantics of office objects is proposed. The model contains features for modeling forms, documents, and other complex objects; these features include a constraint mechanism based on triggers, templates for presenting objects in different media, and unformatted data types such as text and audio. The representation of common office objects is described. User-level commands may be translated to operations within the model.*

Discusses features that need to be added to a specification language for conventional information systems to adapt it to the context of the electronic office. Should be read.

Goguen86

J. A. Goguen. "One, None, a Hundred Thousand Specification Languages." *Proc. IFIP World Congress 1986.* Amsterdam: North-Holland, 1986, 995-1003.

Abstract: *Many different languages have been proposed for specification, verification, and design in computer science; moreover, these languages are based upon many different logical systems. In an attempt to comprehend this diversity, the theory of institutions formalizes the intuitive notion of a "logical system". A number of general linguistic features have been defined "institutionally" and are available for any language based upon a suitable institution. These features include generic modules, module hierarchies, "data constraints" (for data abstraction) and multiplex institutions (for combining multiple logical systems). In addition, institution morphisms support the transfer of results (as well as associated artifacts, such as theorem provers) from one language to another. More generally, institutions are intended to support as much computer science as possible independently of the underlying logical system.*

This viewpoint extends from specification languages to programming languages, where, in addition to the programming-in-the-large features mentioned above, it provides a precise basis for a "wide spectrum" integration of programming and specifications. A logical programming language is one whose statements are sentences in an institution, whose operational semantics is based upon deduction in that institution, giving a "closed world" for a program. This notion encompasses a number of modern programming paradigms, including functional, logic, and object-oriented, and has been useful in unifying these paradigms, by unifying their underlying institutions, as well as in providing them with sophisticated facilities for data abstraction and programming-in-the-large.

A wide-ranging but somewhat technical paper. Contains useful references to Goguen's earlier work on data abstraction and specification languages. The references to OBJ2 are particularly relevant.

Gries81

Gries, D. *The Science of Programming*. New York: Springer-Verlag, 1981.

The first part of this book is an excellent source for material on logic. Gries follows the philosophy that a program and its specification, in the form of assertions, should be developed side by side. This can work very well for programming-in-the-small.

Griswold81

Griswold, R. E., D. R. Hanson, and J. T. Korb. "Generators in Icon." *ACM Trans. Prog. Lang. and Syst. 3* (1981), 144-161.

Abstract: *Icon is a new programming language that includes a goal-directed expression evaluation mechanism. This mechanism is based on generators—expressions that are capable of producing more than one value. If the value produced by a generator does not lead to a successful result, the generator is automatically activated for an alternate value. Generators form an integral part of Icon and can be used anywhere. In addition, they form the basis for the string scanning facility and subsume some of the control expressions found in other languages. Several examples are given.*

This paper introduces the Icon concept of generators and gives examples of their application. It should be examined.

Guttag85

Guttag, J. V., J. J. Horning, and J. M. Wing. "The Larch Family of Specification Languages." *IEEE Software 2*, 5 (Sept. 1985), 24-36.

Abstract: *Larch specifications are two-tiered. Each one has a component written in an algebraic language and another tailored to a programming language.*

An introduction to the specification language Larch. Should be read, at least for an understanding of the two-tiered approach to specification. The two-tiered approach permits Larch to be used for the specification of both abstract data types and data transformers, and, with somewhat less success, even information systems.

Guttag86a

Guttag, J. V., and J. J. Horning. "Report on the Larch Shared Language." *Science of Comp. Programming 6* (1986), 103-134.

Abstract: *Each member of the Larch family of formal specification languages has a component derived from a programming language and another component common to all programming languages. We call the former interface languages, and the latter the Larch Shared Language.*

This paper presents version 1.1 of the Larch Shared Language. It has two major sections. The first part starts with a brief introduction to the Larch Project and the Larch family of languages, and continues with an informal presentation of most of the features of the Larch Shared Language. It concludes with a brief discussion of how we expect Larch Shared Language Specifications to be used, a discussion of some of the more important design decisions, and a summary of the current status of the Larch project. The second part of this paper is a reference manual. A companion paper includes an extensive set of examples.

Larch is a two-tiered specification language consisting of components that are, respectively, independent of and dependent on, the programming language used for the implementation. This paper defines the independent components.

Guttag86b

Guttag, J. V., and J. J. Horning. "A Larch Shared Language Handbook." *Science of Comp. Programming 6* (1986), 135-157.

Abstract: *This handbook consists of a collection of traits written in the Larch Shared Language, and is intended as a companion to the "Report on the Larch Shared Language". It should serve three distinct purposes: Provide a set of components that can be directly incorporated into other specifications; Provide a set of models upon which other specifications can be based; and help people to better understand the Larch Shared Language by providing a set of illustrative examples.*

Companion article to [Guttag86a].

Hayes87

Hayes, I., ed. *Specification Case Studies*. Englewood Cliffs, N. J.: Prentice-Hall, 1987.

A collection of specification case studies expressed in the Z specification language. This language, which is based on set theory, was introduced by Abrial [Abrial80], but is still evolving. The Z notation is very compact, which may detract from readability. A good source of projects.

Heininger80

Heininger, K. L. "Specifying Software Requirements for Computer Systems: New Techniques and their Application." *IEEE Trans. Software Eng. SE-6* (1980), 2-13.

Abstract: *This paper concerns new techniques for making requirements specifications precise, concise, unambiguous, and easy to check for completeness and consistency. The technique is well-suited for complex real-time software systems; they were developed to document the requirements of existing flight software for the Navy's A-7 aircraft. The paper outlines the information that belongs in a requirements document and discusses the objectives behind the techniques. Each technique is described and illustrated with examples from the A-7 document. The purpose of the paper is to introduce the A-7 document as a model of a disciplined approach to requirements specification; the document is*

available to anyone who wishes to see a fully worked-out example of the approach.

This paper emphasizes the specification of performance properties. The techniques are related to the specification of of flight software for the A-7E aircraft.

Henderson86

Henderson, P. "Functional Programming, Formal Specifications, and Rapid Prototyping." *IEEE Trans. Software Eng. SE-12* (1986), 241-250.

Abstract: *Functional programming has enormous potential for reducing the high cost of software development. Because of the simple mathematical basis of functional programming it is easier to design correct programs in a purely functional style than in a traditional imperative style. We argue here that functional programs combine the clarity required for the formal specification of software design with the ability to validate the design by execution. As such they are ideal for rapidly prototyping a design as it is developed. We give an example which is larger than those traditionally used to explain functional programming. We use this example to illustrate a method of software design which efficiently and reliably turns an informal description of requirements into an executable formal specification.*

The intent of Henderson's paper is to support the view that functional programs are their own specifications, and that functional programming should therefore become a primary tool in the software development process.

Hoare78

Hoare, C. A. R. "Communicating Sequential Processes." *Comm. ACM 21* (1978), 666-677.

Abstract: *This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of functional programming exercises.*

A classic paper on parallel programming. Contains interesting examples, including a solution for the dining philosophers problem.

Horning85

Horning, J. J. "Combining Algebraic and Predicative Specifications in Larch." In *Proc. TAPSOFT '85, Vol. 2*. Berlin: Springer-Verlag, 1985, 12-26. Springer-Verlag Lecture Notes in Computer Science, No. 186.

Abstract: *Recently there has been a great deal of theoretical interest in formal specifications. However, there has not been a corresponding increase in their use for software development. Meanwhile, there has been significant convergence among formal specification methods intended for practical use.*

The Larch project is developing tools and techniques intended to aid in the productive use of formal specifications. This talk presents the combination of ideas, both old and new, that we are currently exploring.

One reason why our previous specification methods were not very successful was that we tried to make a single language serve too many purposes. To focus the Larch project, we made some fairly strong assumptions about the problem we were addressing.

Each Larch specification has two parts, written in different languages. Larch interface languages are used to specify program units (e.g., procedures, modules, types). Their semantics is given by translation to predicate calculus. Abstractions appearing in interface specifications are themselves specified algebraically, using the Larch Shared Language.

A series of examples will be used to illustrate the use of the Larch Shared Language and the Larch/CLU interface language. The talk will conclude with notes on the key design choices for each of the languages, and for the method of combining the two parts of a specification.

A fine introduction to the two-tiered specification language Larch, with a discussion of formal specifications in general included as a bonus.

Jardine84

Jardine, D. A., and A. R. Reuber. "Information Semantics and the Conceptual Schema." *Inform. Sys. 9* (1984), 147-156.

Abstract: *The semantics of various proposals for Conceptual Schema languages are compared and contrasted. Concepts are defined using logic and class theory notation, so that terminology is reduced to a common basis. A basis for handling temporal aspects of an Information System is provided.*

A survey of methodologies for the specification of information systems. The authors conclude that temporal effects do not require any special treatment.

Jones78

Jones, C. B. "The Metalanguage: A Reference Manual." In *The Vienna Development Method: The Meta-Language*, D. Bjorner and C. B. Jones, eds.

Berlin: Springer-Verlag, 1978, 218-277. Springer-Verlag Lecture Notes in Computer Science, No. 61.

Abstract: *The recent work of the Vienna Laboratory on the subject of semantic definitions has used the "denotational semantics" approach. Although this is a clear break with the earlier abstract interpreter approach, the newer meta-language has tried to preserve and even improve upon the readability of the earlier "VDL" notation. The meta-language described here has been used in the definitions of large programming languages and systems. This paper is not a tutorial; rather it provides a reference document for the meta-language.*

The reference manual for the Vienna Development Method.

Kemmerer85

Kemmerer, R. A. "Testing Formal Specifications to Detect Design Errors." *IEEE Trans. Software Eng. SE-11* (Jan. 1985), 32-43.

Abstract: *Formal specification and verification techniques are now used to increase the reliability of software systems. However, these approaches sometimes result in specifying systems that cannot be realized or that are not usable. This paper demonstrates why it is necessary to test specifications early in the software life cycle to guarantee a system that meets its critical requirements and that also provides the desired functionality. Definitions to provide the framework for classifying the validity of a functional requirement with respect to a formal specification are also introduced. Finally, the design of two tools for testing formal specifications is discussed.*

Provides operational specifications in terms of an abstract machine for the library example. Examines both rapid prototyping and symbolic execution.

Kowalski85

Kowalski, R. "The relation between logic programming and logic specification." In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1985, 11-27.

Abstract: *Formal logic is widely accepted as a program specification language in computing science. It is ideally suited to the representation of knowledge and the description of problems without regard to the choice of programming language. Its use as a specification language is compatible not only with conventional programming languages but also with programming languages based entirely on logic itself. In this paper I shall investigate the relation that holds when both programs and program specifications are expressed in formal logic. In many cases, when a specification completely defines the relations to be computed, there is no syntactic distinction between specification and program. Moreover the same mechanism that is used to execute logic programs, namely automated deduction, can also be used to execute logic specifications. The only difference between a complete specification and a program is one of efficiency. A program is more efficient than a specification.*

Kowalski argues that a complete logical specification is indistinguishable from a logical program; the only observable difference is one of efficiency. However, most specifications are incomplete in one way or another.

Leveson87

Leveson, N. G., and J. L. Stolzy. "Safety Analysis Using Petri Nets." *IEEE Trans. Software Eng. SE-13* (1987), 386-397.

Abstract: *The application of Time Petri net modeling and analysis techniques to safety-critical real-time systems is explored and procedures described which allow analysis of safety, recoverability, and fault-tolerance.*

Discusses specification of real-time systems in which safety, recoverability, and fault-tolerance have to be provided. A readable paper.

Liskov79

Liskov, B. H., and V. Berzins. "An Appraisal of Program Specifications." In *Research Directions in Software Technology*, P. Wegner, ed. Cambridge, MA: MIT Press, 1979, 276-301. Reprinted in [Gehani86], 3-23.

A survey, to about 1978, of different approaches to the specification of data types and data transformers, with some discussion of parallel programs.

Liskov86

Liskov, B., and J. Guttag. *Abstraction and Specification in Program Development*. New York: McGraw-Hill, 1986.

A textbook on the use of the CLU programming language, which provides facilities for operational specifications in the development of software. There is also a brief introduction to the Larch specification language, which has a more abstract orientation.

Manna81

Manna, Z., and A. Pnueli. "Verification of Concurrent Programs: The Temporal Framework." In *The Correctness Problem in Computer Science*, Boyer, R. S., and J. S. Moore, eds. Academic Press, London, 1981, 215-273.

Abstract: *This is the first in a series of reports describing the application of Temporal Logic to the specification and verification of concurrent programs.*

We first introduce Temporal Logic as a tool for reasoning about sequences of states. Models of concurrent programs based both on transition graphs and on liner-text representations are presented and the notions of concurrent and fair executions are defined.

The general temporal language is then specialized to reason about those execution states and execution sequences that are fair computations of concurrent programs. Subsequently, the language is used to describe properties of concurrent programs.

The set of interesting properties is classified into Invariance (Safety), Eventuality (Liveness) and Precedence (Until) properties. Among the properties studied are: Partial Correctness, Global Invariance, Clean Behavior, Mutual Exclusion, Deadlock Absence, Termination, Total Correctness, Intermittent Assertions, Accessibility, Starvation Freedom, Responsiveness, Safe Liveness, Absence of Unsolicited Response, Fair Responsiveness and Precedence.

In the following reports of this series we use the temporal formalism to develop proof methodologies for proving the properties discussed here.

Although this paper concentrates on verification, the early parts provide a useful introduction to temporal logic and to the issues of functional programming.

Manna85

Manna, Z., and R. Waldinger. *The Logical Basis for Computer Programming, Vol. 1: Deductive Reasoning*. Reading, Mass.: Addison-Wesley, 1985.

A thorough exploration of some basic data types—such as non-negative integers, trees, lists, and sets—as mathematical theories. Although the style is rather dry, this book can be recommended as a gentle introduction to logic as it applies to software specification.

Martin85

Martin, J., and C. McClure. *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, N. J.: Prentice-Hall, 1985.

Diagrams provide an excellent means of communication between the software specifier and the client. However, a badly designed diagram can be a hindrance rather than an aid. This compendium of diagramming techniques and tools emphasizes the difference between good and poor diagrams. This book must be examined.

Matsumoto84

Matsumoto, Y. “Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels.” *IEEE Trans. Software Eng. SE-10* (1984), 502-513.

Abstract: *In the Toshiba software factory, quality control and productivity improvements are primary concerns. Emphasis is placed on reusing existing software modules that have been proven correct through actual operation. To achieve a substantial degree of reuse, the software design process is viewed at several levels of abstraction. In this paper, these levels of abstraction are defined, and examples of the specification for these defined levels are given. This paper proposes a "presentation" of each existing module at the highest level of abstraction. Traceability between the presentation and the reusable program modules which implement it is established to simplify reusability. The paper concludes with an example showing reuse of a presentation for a different application.*

This is ancillary reading. The software development process is examined from the perspective of the software factory, but the adaptation of this material for classroom use would take considerable time.

Naur82

Naur, P. “Formalization in Program Development.” *BIT 22* (1982), 437-453.

Abstract: *The concepts of specification and formalization, as relevant to the development of programs, are introduced and discussed. It is found that certain arguments given for using particular modes of expression in developing and proving programs correct are invalid. As illustration a formalized description of Algol 60 is discussed and found deficient. Emphasis on formalization is shown to have to have harmful effects on program development, such as neglect of informal precision and simple formalizations. A style of specifications using formalizations only to enhance intuitive understandability is recommended.*

A thoughtful critique of too formal an approach to specifications. It is most important to understand that Naur is not opposed to all formalizations, but only to those that obscure meaning. Essential reading to maintain one’s perspective.

Olive86

Olive, A. “A Comparison of the Operational and Deductive Approaches to Conceptual Information Systems Modeling.” *Proc. IFIP World Congress 1986.* Amsterdam: North-Holland, 1986, 91-96.

Abstract: *Conceptual information systems model-*

ing languages can be classified in terms of the approach taken to model the dynamic aspect. Two basic approaches, operational and deductive have emerged up to now. They are characterized in this paper using a common first order logic framework. This provides a basis for comparison and evaluation. Both approaches are then compared in a number of issues. We have found that deductive languages show a number of advantages, which might make further development efforts worthwhile. Aspects requiring research work are also pointed out.

The title conveys the contents very well.

Olle82

Olle, T. W., H. G. Sol, and A. A. Verrijn-Stuart, eds. *Information System Design Methodologies: A Comparative Review*. Amsterdam: North-Holland, 1982.

A collection of papers dealing with the specification and design of an information system for an IFIP Working Conference.

Olle83

Olle, T. W., H. G. Sol, and C. J. Tully, eds. *Information System Design Methodologies: A Feature Analysis*. Amsterdam: North-Holland, 1983.

A continuation of the study of information system specification methodologies begun in [Olle82].

Olle86

Olle, T. W., H. G. Sol, and Verrijn-Stuart, A. A., eds. *Information System Design Methodologies: Improving the Practice*. Amsterdam: North-Holland, 1986.

Another collection of papers dealing with the specification and design of an information system for the IFIP Working Conference.

Parnas72

Parnas, D. L. "On the Criteria to be Used in Decomposing Systems Into Modules." *Comm. ACM 15* (1972), 1053-1058.

Abstract: *This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.*

A classic paper on data abstraction as the basis for modularization. The main example may be dated, but the substance of the paper is not. Essential reading.

Parnas77

D. L. Parnas. "The Use of Precise Specifications in the Development of Software." *Proc. IFIP World Congress 1977.* Amsterdam: North-Holland, 1977, 861-867.

Abstract: *This paper describes the role of formal and precise specifications in the methodological development of software which we know to be correct. The differences between the general use of the word "specification" and the engineering use of that term are discussed. The software development tasks that we are undertaking require a "divide and conquer" approach that can only succeed if we have a precise way of describing the subproblems. It is shown how predicate transformers and abstract specifications can be used when design decisions are made. Two examples of the use of abstract specifications are described and detailed specifications are included.*

Contains a definition of specifications, a list of reasons for having specifications, and a list of reasons for having precise abstract specifications. Essential reading.

Partsch83

Partsch, H., and R. Steinbruggen. "Program Transformation Systems." *ACM Computing Surveys 15* (1983), 199-236.

Abstract: *Interest is increasing in the transformational approach to programming and in mechanical aids for supporting the program development process. Available aids range from simple editor-like devices to rather powerful interactive transformation systems and even to automatic synthesis tools. This paper reviews and classifies transformation systems and is intended to acquaint the reader with the current state of the art and provide a basis for comparing the different approaches. It is also designed to provide easy access to specific details of the various methodologies.*

This survey on the transformational development of programs from specifications should definitely be examined.

Partsch86

Partsch, H. "Transformational Program Development in a Particular Problem Domain." *Science of Comp. Programming 7* (1986), 99-241.

Develops a suite of programs for the application domain of context-free languages by transformation of specifications. The transformations are carried out in the CIP wide-spectrum language (see [Bauer81]). The 110 references are invaluable.

Peterson81

Peterson, J. L. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, N. J.: Prentice-Hall, 1981.

A very readable introduction to Petri nets, with a good selection of examples of their application.

Ramamoorthy78

Ramamoorthy, C. V., and H. H. So. "Software requirements and specifications: status and perspectives." In *Tutorial: Software Methodology*, C. V. Ramamoorthy and R. T. Yeh, eds. Silver Spring, Md.: IEEE Computer Society Press, 1978, 43-164.

Abstract: *This report surveys the techniques, languages and methodologies that have been and are being investigated for the specification of software throughout all phases of development from the early conception stage to the detailed design stage. The vast scope of techniques can only be understood by providing a framework so that they can be categorized. We suggest a classification scheme based on the software system life cycle hoping that the purpose, content and requirements of a particular technique can be justified and evaluated. Summary descriptions of significant individual techniques are included to supplement the overall category description.*

Besides being an inventory of what has been done, the report is intended to provide a perspective of the area. Within our framework, we hope to spot those aspects and problems that have not been addressed adequately and suggest relevant concepts and ideas that may be used to tackle these problems and solve them, ultimately.

An early survey of specification methodologies. Should still be examined, but is less important since the appearance of [Birrell85].

Ramamoorthy86

Ramamoorthy, C. V., V. Garg, and A. Prakash. "Programming in the Large." *IEEE Trans. Software Eng. SE-12* (1986), 769-783.

Abstract: *Ad hoc programming techniques do not work in the development of big software systems. The problems faced in developing large software include starting from fuzzy and incomplete requirements, enforcing a methodology on the developers, coordinating multiple programmers and managers, achieving desired reliability and performance in the system, managing a multitude of resources in a meaningful way, and completing the system within a limited time frame. We look at some of the trends in requirement specification, life cycle modeling, programming environments, design tools, and other software engineering areas for tackling above problems. We suggest several* phase-independent *and* phase-dependent *techniques for programming in the large. It is shown how research in automatic programming, knowledge-based systems, metrics, and programming environments can make a significant difference in our ability to develop large systems.*

An excellent survey of the entire development process of large software systems. Should be read for its emphasis on a proper infrastructure for the development task.

Reisig85

Reisig, W. *Petri Nets: An Introduction*. Berlin: Springer-Verlag, 1985.

An excellent introduction to Petri nets with a good selection of examples of their application. Tends to be more formal than [Peterson81].

Rombach87

H. Dieter Rombach. *Software Specification: A Framework.* Curriculum Module SEI-CM-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Schwartz87

Schwartz, M. D., and N. M. Delisle. "Specifying a Lift Control System with CSP." In *Proc. 4th International Workshop on Software Specification and Design*, Harandi, M. T., ed. Silver Spring, Md.: IEEE Computer Society Press, 1987, 21-27.

Abstract: *CSP is a language and mathematical theory that is well suited for specifying the functional behavior of embedded computer systems applications. Using a lift control system as an example, we illustrate a technique for writing specifications in CSP. We start by formalizing the problem statement as predicates that define the legitimate sequences of events. Next, we define a CSP process that is capable of generating all legitimate sequences of events. This CSP process is an executable model that can be tested and later transformed into an efficient implementation.*

This paper presents a specification of the elevator problem. The main emphasis is on the specification

of a single elevator, but a system of elevators is also briefly discussed.

Shaw81

Shaw, M., ed. *Alphard: Form and Content*. New York: Springer-Verlag, 1981.

A collection of papers dealing with Alphard, a language for specification and programming. The specification of data types in Alphard is in terms of an abstract model.

Specs87

Harandi, M. T., ed. *Proc. 4th International Workshop on Software Specification and Design*. Silver Spring, Md.: IEEE Computer Society Press, 1987.

A collection of case studies that solve four specification problems set by the workshop organizers in advance of the workshop. Three of the problems have been adopted by the SEI as standard examples. Essential reading.

Standard83

ANSI/IEEE Standard 729-1983. "Glossary of Software Engineering Terminology." In *Software Engineering Standards*. New York: The Institute of Electrical and Electronics Engineers, 1984.

The initial definition of *specification* in the module philosophy section is taken from this glossary.

Turner85

Turner, D. A. "Functional Programs as Executable Specifications." In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1985, 29-54.

Abstract: *To write specifications we need to be able to define the data domains in which we are interested, such as numbers, lists, trees and graphs. We also need to be able to define functions over these domains. It is desirable that the notation should be higher order, so that function spaces can themselves be treated as data domains. Finally, given the potential for confusion in specifications involving a large number of data types, it is a practical necessity that there should be a simple syntactic discipline that ensures that only well typed applications of functions can occur.*

A functional programming language with these properties is presented and its use as a specification tool is demonstrated on a series of examples. Although such a notation lacks the power of some imaginable specification languages (for example, in not allow existential quantifiers), it has the advantage that specifications written in it are always executable. The strengths and weaknesses of this approach are discussed, and also the prospects for the use of purely functional languages in production programming.

An exposition of the functional language Miranda, in which recursion equations are combined with some notation from set theory. This set notation allows the functional programming language Miranda also to be regarded as a specification language.

Veloso85

Veloso, P. A. S., and A. L. Furtado. "Towards simpler and yet complete formal specifications." In *Information Systems: Theoretical and Formal Aspects*, A. Sernadas, J. Bubenko, and A. Olive, eds. Amsterdam: North-Holland, 1985, 175-198.

Abstract: *A methodology for the formal specification of data base applications is proposed, which is constructive and leads to simpler and shorter specifications by giving a separate treatment to certain general assumptions.*

Given two states of an information system that is defined in Prolog. A plan generator determines the sequence of events that is to lead from one state to the other.

Wing87

Wing, J. M. "A Larch specification of the library problem." In *Proc. 4th International Workshop on Software Specification and Design*, Harandi, M. T., ed. Silver Spring, Md.: IEEE Computer Society Press, 1987, 34-41.

Abstract: *A claim made by many in the formal specification community is that forcing precision in the early stages of program development can greatly clarify the understanding of a client's problem requirements. We help justify this claim via an example by first walking through a Larch specification of Kemmerer's library problem and then discussing the questions that arose in our process of formalization. Following this process helped reveal mistakes, premature design decisions, ambiguities, and incompleteness in the informal requirements. We also discuss how Larch's two-tiered specification method influenced our modifications to and extrapolations from the requirements.*

An illustration of the two-tiered approach of Larch by means of an example, an information system that specifies the operation of a library.

Yourdon86

Yourdon, E. *Structured Walkthroughs, 3rd ed.*. New York: Yourdon Press, 1986.

A thorough guide to the organization and management of walkthroughs, which are applicable in the validation of specifications.

Zave84

Zave, P. “The Operational Versus the Conventional Approach to Software Development.” *Comm. ACM 27* (1984), 104-118.

Abstract: *The conventional approach to software development is being challenged by new ideas, many of which can be organized into an alternative decision structure called the "operational" approach. The operational approach is explained and compared to the conventional one.*

Argues for problem-oriented specifications that are executable by a suitable interpreter and then transformed into efficient implementations. "Conventional approach" refers to informal, natural language requirements definitions. Zave’s arguments agree with the SF philosophy of specification expressed in [Berztiss86b].

Zave86

Zave, P., and W. Schell. “Salient Features of an Executable Specification Language and its Environment.” *IEEE Trans. Software Eng. SE-12* (1986), 312-325.

Abstract: *This paper presents the executable specification language PAISLey and its environment as a case study in the design of computer languages. It is shown that PAISLey is unusual (and for some features unique) in having the following desirable features: 1) there is both synchronous and asynchronous parallelism free of mutual-exclusion problems, 2) all computations are encapsulated, 3) specifications in the language can be executed no matter how incomplete they are, 4) timing constraints are executable, 5) specifications are organized so that bounded resource consumption can be guaranteed, 6) almost all forms of inconsistency can be detected by automated checking, and 7) a notable degree of simplicity is maintained. Conclusions are drawn concerning the differences between executable specification languages and programming languages, and potential uses for PAISLey are given.*

An introduction to PAISLey, a language for the specification of control systems. PAISLey specifications are operational—they are implementation-independent models of how to solve a problem—rather than non-constructive. This paper should be read for a discussion of the differences between the two types of specifications.