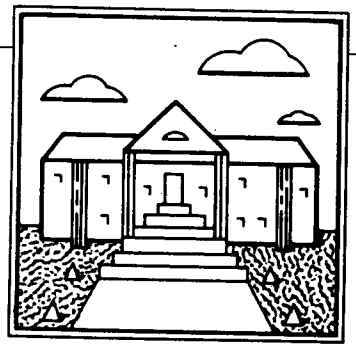
ARTICLE SURVEY



A Study of 12 Specifications of the Library Problem

Jeannette M. Wing, *Carnegie Mellon University*

This study of 12 specifications for a seemingly simple database problem demonstrates many approaches to classify informally stated problem requirements.

The library problem was originally posed seven years ago as a simple exercise in using a formal specification method and language. But its simplicity is deceptive, as 12 specifications proposed to solve the problem show.

This article compares the specifications according to how they addressed problems of the library example to illustrate the imprecision of natural-language specifications and how 12 different approaches to the same set of informal requirements reveal many of the same problems. I admit that the revelation of a problem may be due to the authors' cleverness and not to the particular approach they use; however, each approach undoubtedly helped prod each author into considering certain aspects of the informal requirements — perhaps at the expense of other aspects.

My comparison highlights what issues should be addressed in refining an informal set of requirements and how these issues are resolved in different specification approaches. Thus, for each case, the interesting result of the specification exercise is not the specification itself but the insight gained about the *specificand*. This insight is evidence that benefits can be gained by systematically applying formal, or even informal, specification methods.

History. Richard Kemmerer first posed the library problem in 1981 as part of his formal-specifications class at the University of California at Santa Barbara. He introduced the problem to Susan Gerhart (now with MCC's Software Technology Program) in 1982 when they team-taught an extension class at the University of California at Los Angeles.

In 1984, Gerhart used the problem as a focal point of discussion for the tools group during the second Workshop on Models and Languages for Software Specification and Design.[1] In 1985, Kemmerer's paper on testing formal specifications[2] included a specification of the problem,

0740-7459/88/0700/0066/$01.00 ©1988 IEEE

written in the Ina Jo specification language.

Finally, in 1986, the organizers of the fourth workshop on specifications and design encouraged authors to address a set of four problems, one of which was the library problem, in their position papers. Of the final batch of papers published in the workshop proceedings, 12 addressed the library problem. (The box on p. 80 summarizes the papers.)

This article discusses only those 12, although other library specifications have been written but not formally published. These specifications include Gerhart's written in Affirm,[3] Martin Feather's written in Gist,[4] and Steve King and Ib Sørenson's written in Z.[5]

Problem definition. Exactly what is the library problem? In the call for papers[6] that resulted in the 12 specifications, it was:

"Consider a small library database with the following transactions:

"1. Check out a copy of a book. Return a copy of a book.

"2. Add a copy of a book to the library. Remove a copy of a book from the library.

"3. Get the list of books by a particular author or in a particular subject area.

"4. Find out the list of books currently checked out by a particular borrower.

"5. Find out what borrower last checked out a particular copy of a book.

"There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

"• All copies in the library must be available for checkout or be checked out.

"• No copy of the book may be both available and checked out at the same time.

"• A borrower may not have more than a predefined number of books checked out at one time."

This definition was different from Kemmerer's original statement in several ways:

- The original asked you to consider a "university library database."
- The original did not restrict transaction 1, checkout and return, to only staff users.
- The original had a fourth constraint: A borrower may not have more than one copy of the same book checked out at one time.

Some of the 12 papers' authors were familiar with Kemmerer's original statement, which naturally added to the variation among the interpretations of the problem.

The authors of most of the papers used the library problem to exemplify their points of view about software specification.

Specification approaches

The intention of most of the 12 papers was not to give a straightforward specification of the library problem but to use it to exemplify an author's point of view about software specification. Authors of those papers used the library problem to make their points more concrete. Some papers used the library problem to illustrate a particular specification method or language, but even in some of those papers, the authors gave only fragments of a larger specification.

Because some of the papers do not intend to give explicit specifications, let alone complete ones, I cannot make a true comparison of the specifications. I also do not compare papers because they cover a wide variety of topics, ranging from the use of domain knowledge for writing specifications to the formal definitions of their sufficiency and completeness. You should refer to the individual papers for more information about the different authors' views on software specification and design.

However, all the authors wrote of the problems they encountered with the library example while applying their specification method or language to it in an effort to refine the informal requirements. Thus, this article compares the 12 papers according to how the different specification methods or languages gave the authors insight into the library problem.

I distinguish between the specificand — that which is being specified — and the specification — the text that describes the specificand. Here, I take the specificand to be the behavior of a library as informally conveyed by the problem definition in the call for papers.

Table 1 summarizes the papers according to their level of formality, which phases of the software life cycle the authors address, the papers' key ideas, and the concrete specification language used, if any. In all the tables in this article, the information on the first four papers is shaded in gray to indicate they used informal approaches; the other eight used formal approaches.

Formality. A specification is formal if it has a precise and unambiguous semantics. A precise and unambiguous semantics is given by mathematics usually in the form of a set of definitions, a set of logical formulas, or an abstract model. These three approaches to giving formal semantics roughly correspond to the denotational, axiomatic, and operational approaches of giving semantics to programs.

If an abstract model of the specification

Table 1.
Summary of the papers.
(The papers shaded in gray in this and subsequent tables used informal approaches; the others used formal approaches.)

Paper	Formality	Life-cycle phase	Key idea	Language/project
Kerth	informal	requirements through informal specification	structured analysis and human interface	—
Fickas	AI	requirements through formal specification	usage scenarios	Kate
Rich et al.	AI	requirements through formal specification	Requirements Apprentice	Programmer's Apprentice
Lubars	AI	specification through design	design schema	—
Dubois/ van Lamsweerde	logic	requirements through formal specification	metaspecification	—
Wing	logic	requirements through formal specification	benefit of formalism	Larch
Rudnicki	logic	analysis	testing and proving	Ina Jo
Yue	logic	analysis	sufficient completeness and pertinence	Gist
Levy et al.	logic and executable	requirements through formal specification	multiple methods	Sasco
Terwilliger/ Campbell	logic and executable	design through code	Anna and Prolog	Please
Lee/Sluizer	executable	requirements through formal specification	models of behavior	SXL
Rueher	executable	design through code	rapid prototyping and graphical Prolog	Prolog

is a machine-executable interpreter (like a Prolog interpreter), I consider the specification to be executable. Less obvious is that a specification in logic may also be executable, although its execution may involve the search of a solution space and hence may be infeasible for nontrivial examples, as the original workshop discussion cautioned.[1] In this article, I do not classify such specifications as executable.

A formal specification is usually written in a concrete language, which is like adding syntactic sugar on top of mathematics. If this language has a precisely defined syntax and semantics, a specification written in it is formal. Informality is introduced by the reliance on English and uninterpreted diagrams when writing and giving the specification semantics.

Of the four informal specifications presented in the papers, three use application-domain knowledge (indicated as "AI" in Table 1). All four are presented with tool support in mind so that some amount of machine processing, like pattern-matching on keywords, could be performed on the specifications.

The underlying basis for all but two (Lee/Sluizer and Rueher) of the formal specifications is full, first-order predicate logic. These specifications use preconditions and postconditions to specify state transitions; they use other assertions, ranging from algebraic equations to first-order logical formulas, to specify state invariants and other system constraints. Postconditions refer to both initial and final states of an object. Some specifications are explicit as to which objects are not modified in a state transition.

Levy et al. use a Lisp interpreter to execute their specifications, performing only partial evaluation for incompletely defined functions. The three other executable specifications used Prolog.

The Terwilliger/Campbell technique starts with a logic-based language but restricts the generated specifications to Prolog, resulting in an executable Prolog implementation. They acknowledged the loss of logical power because, in pure Horn clause programming, there is no way to specify explicitly that a formula is false. Prolog's solution is to use negation-as-failure under the closed-world assumption: A fact that is not provably true is assumed to be false.

Lee and Sluizer chose Prolog as simply the implementation language of their specification language, SXL. While Prolog and SXL have some features in common, they are entirely separate languages with different semantics. For example, SXL has no concept of backtracking, which is fundamental in Prolog; SXL uses forward-chaining rules, which are absent in Prolog.

Rueher uses Prolog as both an implementation and specification language. Thus, his logic was restricted to Horn clauses and was further corrupted by his indirect use of assert and retract predicates, which modify the initial set of Prolog facts.

The virtues of formal specifications have been argued elsewhere.[7] I have also argued their virtues for the library problem.[8]

Life-cycle phase. Most of the authors assumed a traditional software life-cycle development process broken into several phases:

- stating informal requirements,
- writing a specification (informal or formal or both),
- developing a design,
- writing the code, and

• testing, validating, or verifying the code.

They did vary their emphases, however:

• One paper (Kerth) focused on elaborating informal requirements; I consider "elaborated requirements" to be an informal specification.

• Six papers addressed the transition from informal requirements to formal specifications; however, two (Fickas and Rich et al.) stopped short of giving formal specifications.

• One paper (Lubars) focused on the transition from informal specifications to design.

• Two papers (Terwilliger/Campbell and Rueher) focused on the transition from design to code.

• Two papers (Rudnicki and Yue) proposed analyzing the specification itself, both during and after its development.

• The three papers that used Prolog (Terwilliger/Campbell, Lee/Sluizer, and Rueher) advocated rapid prototyping and regarded an executable specification as a prototype.

• Two papers (Dubois/van Lamsweerde and Levy et al.) addressed the specification process itself and how to specify the actions, such as refinement and abstraction, performed during specification.

The more a specification looks like code, the easier it might be for a programmer to transform it into an efficient implementation or verify an implementation. On the other hand, the specification might be harder for a nonprogrammer to read, and it very likely would contain premature design decisions. In fact, Fickas and Rich et al. were concerned with how to make the specification resemble the informal requirements — *not* how to make it resemble code.

Specification details

I distinguish between a specification method and the specification itself, that piece of text that results from following a method. Table 2 categorizes the papers' specifications along three lines:

• whether the specification is operation-oriented or data-oriented (object-oriented),

• whether it is composed of separate modules, and

• whether graphics or diagrams are used to aid readability.

The first two criteria are purely subjective.

Orientation. Specification methods, which mimic traditional programming methods, tend to focus on either a system's operations or its data.

An operation-oriented method identifies the system's key functions and describes each function's I/O behavior. Data might be treated as globally accessible, and their behavior is either left unspecified or described through simplistic models; initial conditions are typically stated explicitly.

A data-oriented method identifies the key types of data that the system manipulates and describes an object's behavior by describing the operations that may access it. The system's functionality is either left unspecified or indirectly described through the interactions of the operations of the different data types; initial conditions are typically stated implicitly through create operations on objects.

For the library example, a typical operation-oriented method would describe the I/O behavior of each transaction (such as check out a copy and add a copy) but might leave implicit the books' properties. A data-oriented method would describe what libraries, books, and users are, but it might leave implicit the library system's overall I/O behavior.

Table 2 shows that three specifications are not just operation-oriented. Dubois/van Lamsweerde paid equal attention to both data and operations, while Wing and Rueher both focused on data. Dubois/van Lamsweerde and Wing specified the semantics of data in terms of algebraic abstract data types. Because Rueher's ultimate concern was to transform a high-level design into code, he generated from his Prolog specification a high-level design consisting of Ada package definitions for types; data semantics were still in terms of Prolog predicates.

The other nine specifications focused more on specifying the effects of the operations, not on the properties of the data. Of course, this orientation could be attributed to the operational (transactional) presentation of the call for papers and not to the orientation of the authors' specification method.

In fact, a specification method's orientation may be different from the resulting specification's orientation. For example, contrary to what Table 2 might imply, two other specification methods could be used

Table 2.
Summary of the specifications.

Paper	Orientation	Modularity	Readability
Kerth	operation	—	graphics
Fickas	operation	—	—
Rich et al.	operation	—	—
Lubars	operation	—	schemata
Dubois/ van Lamsweerde	both	yes	—
Wing	data	yes	Venn diagrams
Rudnicki	operation	—	—
Yue	operation	—	—
Levy et al.	operation	yes	—
Terwilliger/ Campbell	operation	yes	—
Lee/Sluizer	operation	—	—
Rueher	data	—	graphics

A glimpse at the 12 papers

The 12 papers were published in *Proceedings of the Fourth International Workshop on Software Specification and Design* (CS Press, Los Alamitos, Calif., 1987).

Kerth. For specifying real-life systems, Norman Kerth noted in "The Use of Multiple Specification Methodologies on a Single System" (pp. 183-189) the inadequacy of the structured analysis design method as described by Tom DeMarco[1] and augments it with a three-dimensional Human Interface Perspective, which contains

- a collection of graphical views that look roughly like screen menus from which you select what function to do next,
- a view-transition diagram that represents the control flow as you move from one view to another as different functions are selected, and
- a textual description of the user interface's behavior, including what normal or undesirable events occur when keystrokes are entered or the mouse is clicked on options.

Fickas. In "Automating the Analysis Process: An Example" (pp. 58-67), Stephen Fikas described the components of a system called Kate that helps automate the transformation of informal requirements into a formal specification. The paper showed how to criticize refinements of informally stated requirements through domain knowledge, usage scenarios, and intermediate summaries. Fickas also relied on human experts (professional librarians) to help critique a specification and make Kate smarter by enlarging its knowledge database.

Rich et al. In the context of the ongoing Programmer's Apprentice research project at the Massachusetts Institute of Technology, Charles Rich, Richard Waters, and Howard Reubenstein described in "Toward a Requirements Apprentice" (pp. 79-86) the use of a requirements apprentice to help a user convert an initial informal requirement into a formal specification.

The assistant relies on simple deductive methods applied to extensive domain knowledge represented as clichés. For example, the library example uses clichés about repositories (where objects like books are stored), information systems (programs for storing and reporting data), and tracking systems (programs for keeping track of the current state of objects like the physical library repository).

Lubars. In "Schematic Techniques for High-Level Support of Software Specification and Design" (pp. 68-75), Mitchell Lubars addressed design reusability by defining abstract graph representations of designs and then instantiating them to yield specific designs. He instantiated an inventory-control scheme with domain knowledge (library databases) to model the library.

Dubois/van Lamsweerde. In "Making Specification Processes Explicit" (pp. 169-177), Eric Dubois and Axel van Lamsweerde addressed how to specify the process of specifying. Based on a dual object model and operation model of specification, they suggested two metamodels where the specification process is made explicit: The process model captures the steps used by a specifier while constructing a specification; the method model captures the control information and rationale used for the steps taken, yielding an overall specification strategy.

Wing. In "A Larch Specification of the Library Problem" (pp. 34-41), I demonstrated the benefits of formal specifications by identifying many problems with informal statements of requirements such as in the library example. I presented a specification of the library using Larch, which combines algebraic and predicative specification techniques into one framework.

Rudnicki. In "What Should Be Proved and Tested Symbolically in Formal Specifications?" (pp. 190-195), Piotr Rudnicki argued that errors in specifications be detected by both testing and proving properties about them *by hand* (his emphasis). Each test case is related to some property that might best be simulated through symbolic execution of the specification or might more easily be provable from the specification itself, he said. His specification is based on Kemmerer's Ina Jo specification.

Yue. In "What Does It Mean to Say that a Specification Is Complete?" (pp. 42-49), Kaizhi Yue formally defined two properties — sufficient completeness and pertinence of a set of goals — that capture the notion of whether a specification contains enough, but no more than necessary, information to achieve the goals. He described how to analyze a specification in terms of these properties, using a Gist specification of the library problem as an example.

Levy et al. Like Dubois and van Lamsweerde, Nicole Levy, A. Piganiol, and J. Souquieres were interested in the specification process itself. In "Specifying with Sasco" (pp. 236-241), they briefly describe Sasco, a system to support the evolution of an informal description into a formal specification. Through operations like refinement, enrichment, reuse, and abstraction, Sasco supports a multimethod approach to specification, where "method" means a description of the process of specifying. So, instead of being limited to just one approach like top-down or data-oriented, Sasco supports several specification approaches.

Terwilliger/Campbell. In "Please: A Language for Incremental Software Development" (pp. 248-256), Robert Terwilliger and Roy Campbell presented Please, an executable specification language that combines Anna (annotated Ada) and Prolog. Please models data with Ada types and specifies transactions with both Ada procedures and annotations that define preconditions and postconditions and auxiliary predicates. Preconditions and postconditions are written in terms of Horn clauses, so they can be directly translated into Prolog and executed.

Lee/Sluizer. In "SXL: An Executable Specification Language," Stanley Lee and Suzanne Sluizer asserted that building and analyzing models as practiced in traditional engineering disciplines should be done in software engineering as well. They presented SXL, an executable specification language based on a state-transition model. SXL uses transition rules via preconditions and postconditions to specify allowed behavior; it uses logical invariants to specify required behavior. The invariants are automatically enforced during model execution.

Rueher. In "From Specification to Design: An Approach Based on Rapid Prototyping" (pp.126-133), Michel Rueher presented a graphical syntax for Prolog, which is used as an executable specification language. He endorsed rapid prototyping through executable specifications, obtained by using Prolog as both his specification and implementation language.

Reference

1. T. DeMarco, *Structured Analysis and Systems Specification*, Yourdon Press, New York, 1979.

to generate specifications that are just data-oriented or both operation-oriented and data-oriented. Levy et al. support a multimethod approach where the user is free to choose one method or design his own. And, although the specification in Terwilliger/Campbell focused only on the operations of interest, their system's use of Anna, and hence Ada, could be used in a data-oriented manner.

For large, complex, and realistic systems, favoring one orientation over the other is too naive. A dual specification method (like in Dubois/van Lamsweerde) or, more generally, a multimethod approach (like in Levy et al.) is more appropriate. But giving a formal meaning to a specification that results from a mix of methods remains a challenge for researchers.

Modularity. For programs and large-system design, the benefits of modularity — such as increased modifiability, reusability, and readability — are well-known. For specifications, they are equally important. Various modularity techniques — such as inheritance, type abstraction, and parameterization — that are commonly used to structure programs today are equally useful for building specifications. Moreover, a modular, formal specification can be used to do proofs in pieces and thus help isolate parts of the requirements that are responsible for certain design decisions.

On the other hand, a method that supports modularity is not a panacea: First, as with any method, it is possible to misuse it. Second, it may be hard to avoid spreading some design decisions throughout an entire specification, like deciding that only one staff person may be hired to run a library.

Table 2 indicates which formal-specification approaches explicitly support modular construction of specifications. A dash for a formal specification indicates that a paper used a single specification to describe the entire system's behavior. For the informal-specification papers, not enough information was provided for me to determine whether any of the informal techniques explicitly supports modularity.

Graphics. Kerth's approach used graphics to specify the interface between a user and the library system. Lubars used schematic diagrams (labeled circles and labeled arrows) to describe different kinds of database systems and their instances. Wing used Venn diagrams to illustrate the subset relations between different kinds of users and between different kinds of books. Rueher added a graphical syntax for Prolog with the intent of improving not only the readability but also the debugging of a Prolog program.

Kerth's use of graphics especially helped me understand his model of the library problem. He presented a collection of views of Apple Macintosh-like screen displays at different levels of the system hierarchy to simulate what a user would see on a terminal. He also used a state-transition diagram to depict the flow from view to view as different transactions are performed.

Comparison

The process of refining the informal requirements of the library example toward a more detailed informal specification or toward a formal specification revealed different kinds of problems with informal descriptions, such as inconsistencies, oversights, ambiguities, and incompletenesses. Here, I consider only ambiguities and incompletenesses.

Ambiguities. Table 3 summarizes how each paper treated the five major ambiguities I found. A dash for a paper that presented a formal specification indicates

Table 3.
Ambiguities.

Paper	Library	User	Book	Available	Last checked out
Kerth	yes	=2	book ≠ copy	—	—
Fickas	—	>2	—	redundant	last = current
Rich et al.	yes	—	book ≠ copy	—	—
Lubars	yes	—	—	—	last = current
Dubois/ van Lamsweerde	yes	>2	book ≠ copy	implies in library	—
Wing	—	>2	book ≠ copy	implies in library	last = current
Rudnicki	—	=2	book ≠ copy	redundant	—
Yue	yes	—	book ≠ copy	—	—
Levy et al.	yes	=2	book ≠ copy	implies in library	—
Terwilliger/ Campbell	—	=2	book = copy	implies in library	last = current
Lee/Sluizer	yes	=2	book ≠ copy	implies in library	last ≠ current
Rueher	yes	=2	book = copy	implies in library	last ≠ current

that insufficient information was given for me to make an interpretation. Entries for informal specifications indicate what an author explicitly stated in the paper. While I could infer information from a formal specification and often did not rely on what the authors said explicitly in text, I chose not to try to infer from informal specifications.

The five major ambiguities were:

1. *What is a library?* Table 3 indicates with "yes" which authors explicitly distinguished a library database from the entire library system. A library database includes records of books (like author and title, and perhaps copy number) and records of users (like name and status). Transactions are performed on the database explicitly by some implicit set of users. An entire library system includes not only a library database (also called "inventory," "repository," and "card catalog") but also the people using the library, the books on the shelves, and the transactions involving all these objects.

The distinction between a library database and a library system arises from deciding what part of the library concept is part of the specificand (the library) and what is part of the specificand's environment.

If the library is just the *database*, the environment must include the people who have access to the database (those who perform transactions on it).

If the library is the entire *system*, including the people and books, the environment of the database becomes a part of the library system itself; the library's environment would then be the rest of the university (if a university library) or perhaps other public services (if a public library).

Though some of the authors described this ambiguity, none of the specifications made clear the distinction between the specificand and its environment. In fact, the Gist specification language[4] — which Yue used — models what it calls "closed systems," intentionally blurring the distinction between a system and its immediate environment.

2. *What is a user?* Most specifications assumed that library users are divided into two disjoint classes: those with the privileged library-staff status and those without (ordinary borrowers). Table 3 indicates with "=2" which specifications divided users into exactly two classes.

There are, however, other reasonable interpretations of what a user is:

- Assuming a distinction between a library system and a library database, one interpretation (in Dubois/van Lamsweerde) is that a system user may be different from a database user. In this case, a database user is not necessarily the same as a person with staff status because an ordinary borrower might use the database to find out what books he has checked out.
- Wing distinguished between someone affiliated with the library and someone who is not; only those affiliated with the library can have both staff and ordinary status, depending on the action they take.
- A third interpretation (in Fickas) is based on the original problem definition, which qualified the library to be a university library database. Fickas reported that a professional librarian interpreted "users" to mean ordinary borrowers and "staff" to mean organizational staff: university staff as opposed to university faculty or students. In this case, staff people would presumably be more privileged than professors.

Most specifications assumed that library users are divided into two disjoint classes: those with privileged status and those without.

A consistent interpretation of both "library" and "user" is crucial to interpret the restriction that transaction 1 (check out and return a book) be performed by staff users:

- A straightforward interpretation of the restriction is that only staff members may check out or return a book and that ordinary borrowers may not do either. This interpretation leads to the unreasonable situation where ordinary borrowers can find out what books they have checked out but can never have checked out any books. An answer to a "list checked out books" query would always result in an empty list!
- A more reasonable interpretation is that only staff members may check out or return a book but that they do so *on behalf* of ordinary borrowers. Here, three objects are involved: the book, the staff member, and the borrower.
- A third possibility exists: A careful reading of Kemmerer's original problem definition led me to believe that this restriction was an unintentional mistake in the call for papers.

3. *What is a book?* Most specifications distinguish carefully between a book and a copy of a book. A book might be modeled as having an author and a title (and perhaps, a subject). Copies of a book are assumed to be physically distinct from one another and thus are uniquely identifiable. Copies with the same author and title are then considered to be the same book.

Table 3 indicates whether the specification associates a unique identifier with the idea of a "book," thus equating the notions of "book" and "copy" or whether it associates a unique identifier with the idea of a "copy," thus modeling a book as a set of copies. In two cases (in Lee/Suizer and Wing), the unique identifier is not explicitly modeled but assumed to be associated with each physical entity in the system.

Deciding what "book" means affects the meaning of the rest of the specification. For example, what is returned by transaction 4 (list books), which refers to "books," not "copies of books"? The statement of transaction 4 could have been sloppily written because if it had said "copies of books," it would be consistent with other uses of "copy of book." If so, the term "book" instead of "copy of book" would have sufficed and books would be uniquely identifiable. On the other hand, perhaps the transaction is not meant to distinguish between the numerous copies a borrower may have of the same book.

Another possibility exists: The call for papers lacked Kemmerer's original fourth constraint, which said that a user may have only one copy of a book checked out at any time. This constraint is consistent with the informal statement of transaction 4 in the call for papers because it would make it clear that the transaction need not be concerned with returning copy numbers as

well as book identities (author and title).

4. *What does "available" mean?* Two papers (Rudnicki and Fickas) consider constraints 1 and 2 as stating the same thing (indicated as "redundant" in Table 3). A book is either available or checked out; it cannot be both. (The constraints were that all copies of a book be available or be checked out and that no copy of a book be simultaneously available and checked out.)

Other authors (see "implies in library" in Table 3), however, distinguish not only whether a book is available but whether it is even associated with the library at all (it could be in a bookstore or privately owned). Thus, if a book is available (or checked out), it must be associated with the library. There may be books not associated with the library that are neither available nor checked out. This interpretation is consistent with constraints 1 and 2, yet they do not cause one to be a restatement of the other.

As an aside, Fickas noted that besides being available or checked out, there are other states, such as being lost or stolen, that a library book might be in.

5. *What does "last checked out" mean?* Fickas's professional librarian noted that of the books on the shelves it is not interesting to find out who last borrowed them, so "last" must mean "currently." Authors of three other papers also equate the notion of "last checked out" with "currently checked out," although a distinction is implied by the difference in wording between transactions 4 and 5 (list books and find out who last checked out a book). Equating the notions means that transaction 5 returns a current borrower.

Lee and Sluizer, however, interpreted "last checked out" to be different from "currently checked out" by making the set of books currently checked out a subset of books that are last checked out: If someone currently has a book checked out, that person must also be the last person to have checked out the book. In Lee/Sluizer, transaction 5 returns either the current borrower if the book is checked out or the last borrower if the book is not checked out.

Rueher also interpreted "last checked out" to be different from "currently checked out." His paper's transaction 5, however, faithfully reflects the informal specification and returns the last borrower of only available books (and no current borrowers).

Incompletenesses. There are many kinds of incompletenesses in the informal requirements. Table 4 summarizes the six major incompleteness categories. I do not address undefined terms like "title" and "subject," which could also be classified as a kind of incompleteness.

1. *Initialization.* As Table 4 indicates, three papers explicitly characterized what properties must hold in the initial state of the system. In the state-transition models in Rudnicki and Lee/Sluizer, properties that must hold in the initial state are explicitly written in the specification. Lee/Sluizer specified that initially there exists a normal user, a staff user, an available book, and the book's entry in a card catalog (the library database). Rudnicki specified that the library starts out with no books, no user has any books, and all books have the status of being not checked out. Rudnicki further proved from his specification that to start any interesting

Table 4.
Incompletenesses.

Paper	Initialization	Missing operations	Error handling	Missing constraints	Change of state	Nonfunctional
Berth	—	—	error	—	—	human factors
Fickas	—	—	signals	—	—	system
Rich et al.	—	yes	pre + error	—	—	—
Lubars	—	—	pre	—	update records	liveness
Dubois/ van Lamsweerde	—	—	signals	yes	—	system
Wing	yes	yes	pre + signals	yes	explicit change only	—
Rudnicki	yes	—	pre	yes	eplicit no change	system
Yue	—	—	pre	—	—	liveness
Levy et al.	—	yes	pre	no	explicit change only	liveness
Terwilliger/ Campbell	—	—	pre	no	—	—
Lee/Sluizer	yes	yes	pre	yes	implicit no change	system
Rueher	—	—	error	no	—	programmer interface

activity in the library system, a user of staff status must exist. That person can then add a book to the library, which can then be checked out, later returned, and so on.

Wing included a library-create operation that establishes the initial condition as stated in Rudnicki's specification.

2. *Missing operations.* Some papers noted the inadequacy of the given set of transactions and proposed adding some missing operations. For example, two operations would be useful if a distinction between a book and a copy is made: (1) Add a new book, as opposed to a *copy* of a book (in Levy et al. and Lee/Sluizer), and (2) remove all copies, as opposed to a single copy, of a book (in Rich et al.) and thus remove the existence of a book (a set of copies).

Two operations would be needed to establish a state from which library activity can begin: (1) Create a library and (2) add a staff user (both in Wing).

Two operations are strictly not necessary: (1) Add a regular user and (2) remove a user (both in Wing). But without the first, there would be no need to distinguish between two types of users if the operation of adding a staff user is included. Including the second makes the set of transactions more closely reflect reality and more symmetrical if operations to add users are included.

3. *Error handling.* The informal requirements do not state what should happen if an error is encountered, like trying to return a book that has not been checked out. A specification could either strengthen the precondition of a transaction to prevent the error from arising or strengthen the postcondition by explicitly specifying behaviors for the exceptional cases.

In strengthening the postcondition, you could use a single, catchall error or treat each exceptional case individually. Table 4 indicates whether the specification handles errors by strengthening only the precondition ("pre"), using a single catchall error ("error"), tuning error handling for different situations ("signals"), or some combination of preconditions and error handling.

Some of the errors that the authors addressed include:

- Checkout: Make sure the book being checked out is not already checked out (in Kerth, Lubars, Wing, Terwilliger/Campbell, Lee/Sluizer, and Rueher). Make sure the book is part of the set of library books (in Rich et al., Wing, Terwilliger/Campbell, and Lee/Sluizer).
- Return: Make sure the book is checked out by the user returning the book (in Wing, Rudnicki, and Rueher). Here, you could argue that this is not necessarily an error because it may not matter who returns a book, just as long as it is returned.
- Add book: Make sure the book does not exist (in Kerth, Yue, and Rueher). If a distinction is made between a book and a copy, then adding a copy should check to see if the book exists (in Levy et al. and Lee/Sluizer) or it should explicitly state that a new entity is added (in Wing and Lee/Sluizer).
- Remove book: Make sure the book exists (in Kerth) or is available, implying that it exists (in Wing, Terwilliger/Campbell, Lee/Sluizer, and Rueher).

The informal requirements do not state what should happen if an error is encountered, like trying to return a book that hasn't been checked out.

Finally, for completeness, specifications should treat type errors. If an argument or result of an operation is of the wrong type, the specification contains an inconsistency. All the methods do implicit type checking through the declarations of the types of an operation's arguments and results. This type information is implicitly conjoined to the preconditions and postconditions of individual operations or defined in the underlying semantics by using predicate logic with typed variables.

4. *Missing constraints.* In an informal sense, all authors added more constraints to the three in the call for papers, simply by informally elaborating the requirements or making them more precise. The domain-knowledge papers added domain-specific constraints by, for instance, introducing knowledge about information-retrieval systems for which a library is a special instance.

In a more formal sense, however, you can define a constraint to be a state invariant to be maintained across state transitions in the execution of individual transactions. For those with formal specifications, Table 4 indicates which authors explicitly added more constraints to the call for papers. Examples of constraints that were added are:

- A borrower may not have more than one copy of the same book checked out at a time (in Wing and Rudnicki).
- There is a one-to-many relation between a book and its copies in the library (in Dubois/van Lamsweerde and Lee/Sluizer).
- There is a one-to-one relation between a book and its last borrower (in Lee/Sluizer).

For the latter two constraints, Lee/Sluizer guarantees that, to maintain the invariant, an object (such as a card-catalog entry) is automatically deleted when necessary (such as when a book is removed from the library).

In a formal specification, state invariants like the above constraints are typically specified as a separate global condition and implicitly conjoined to the preconditions and postconditions of each operation. Such global invariants are often discovered while specifying an individual operation. For example, an implicit (unmodified) precondition may be found to be just an instance of a more general invariant, or a specific error case can more generally be subsumed by an invariant.

Conversely, you can distribute an invariant to just the pertinent operations by affixing the appropriate precondition and postcondition. For example, the call for paper's third constraint, which limited how many books a borrower may have, showed up explicitly in the specification of the checkout operation but not in the add or remove operations. Thus, it is often difficult to determine whether a constraint is missing in general or whether an operation's precondition or postcondition must be strengthened.

5. *Specifying change of state.* Some specifications were precise as to what objects change from state to state.

In the Ina Jo specification, as used by

both Rudnicki and Kemmerer, the explicit *NC'(x)* assertion says that *x*'s value does not change from the current state to the next.

Lee/Sluizer said that "each postcondition includes only changes to the system state: Values that are not explicitly mentioned are unchanged." In the Larch specification (as used in Wing), a *modifies* $[x_1, \ldots, x_n]$ clause states that the only objects whose values may change are those in the set $\{x_1, \ldots, x_n\}$.

Lubars mentioned the notion of explicit "update records," which implies that all changes in state must be recorded explicitly. With a way to state explicitly what has changed or to state implicitly what has not, you can make precise a constraint like "The responsibility of a user for a book changes when the user checks out or returns the book" (in Wing, Rudnicki, and Lee/Sluizer).

6. *Specifying nonfunctional behavior.* The papers explicitly addressed three kinds of nonfunctional behavior: human interaction with the library system, system constraints, and liveness.* Only one paper (Kerth) addressed human factors by describing a menu-driven interface to depict how people would interact with the system.

Two system constraints left unspecified are the user's borrowing limit and the library's size. Many authors introduced an uninterpreted variable like "max" or "limit" to denote the borrowing limit but placed no further constraints on it. Dubois/van Lamsweerde, however, constrained it to be nonnegative. Rudnicki moreover constrained it to be positive, not simply nonnegative — a restriction he discovered when testing his specification. Lee/Sluizer actually provided a limit (five books) to make the specification concrete and fully executable.

Fickas described at length the implication of removing the borrowing-limit constraint. For example, he said that placing a borrowing limit may prevent a user who needs more books than allowed from achieving his goal. He also questioned what "small" means. "Small library database" (as specified in the call for papers) could mean a small-library database, a small library-database, a small-time system, or a simple problem involving a library database, he said.

Three papers described progress as a desired liveness property of the library system. Lubars assumed that the class of inventory system he instantiated to get a library system is one for which goods (books) are returned as opposed to one for which they are not (like food).

Yue explored the constraint on borrowers even further. He argued that progress could be impeded if either of two situations arises:

- A user wants to check out a book and has a maximum number already. He is forced to return a book first.
- A user wants to check out a book, but it is not available because someone else has checked it out.

The library problem is deceptively simple because real libraries are not simple and a specification that works for a small library may not apply to a large one.

To solve the second, the library could simply keep adding books — an unrealistic solution. Yue solved both problems at once by adding the constraint that a borrower may not keep a book forever, later refining "forever" to "a predefined period of time."

Dubois/van Lamsweerde did not describe liveness explicitly, but it did introduce enough formalism (a sequence of times, in particular) so that it could characterize liveness properties. For instance, it used these times to determine whether a book had been returned by checking to see that each returned date associated with the book is less than the last checkout date.

Observations

I chose only a subset of the ambiguities and incompletenesses in the informal statement because those chosen represent the ones brought out most often by all 12 papers. The ones I did not present would have revealed nothing further about one specification method over another. The statement of the library problem is deceptively simple. On one hand, its simplicity lends itself nicely to illustrating different specification methods and languages in the length of a workshop paper (six to eight pages). It is simple enough for one person to understand and it is easy to explain to others. For instance, in explaining the pitfalls of informal requirements, I can appeal to your intuition about libraries. On the other hand, its simplicity is deceptive in two ways:

- Real libraries are not simple. They involve more than just people, books, and a database. They have policies according to who the borrower is, what kind of book it is, what time of year it is, and, of course, exceptions to all these policies.
- Just because a specification method or language can be applied to a small library does not mean it can scale up and be applied to a larger system or to a more complicated one. This second deception is compounded by the first if you do not have the luxury to go to experts for their advice, the time to research the literature, or the opportunity to examine existing systems.[9]

None of the specifications addressed in any detail, if at all, many realistic system properties, including:

- Concurrency: What happens if two checkout transactions occur simultaneously?
- Reliability: What happens if the library database-management system crashes?
- Fault-tolerance: What happens if a borrower decides to abort in the middle of a transaction?
- Security: How does the system authenticate the identity of borrowers?

Other issues that arise in any realistic database-management system must also be considered in a more complete specification, including version control, report generation, and managerial policies.

Finally, two broad classes of specification approaches are missing from those presented in the 12 papers. First, although the problem is about a database, no database approach such as semantic data modeling is represented. Second, except for struc-

*Liveness is the quality of making sure something good happens (as opposed to safety, which is the quality of making sure nothing bad happens). Liveness ensures, for example, that a calculation's results are returned (safety ensures only that the calculation is performed and that is performed accurately, not that it is used).

tured analysis (in Kerth), no commercially used method such as Jackson's Structured Programming[10] is represented.

These other approaches are probably missing because their users did not participate in the workshop and because of the program committee's biases. The workshop draws together participants who have a common interest in software specification and design and who tend to take an academic view of the software-design process. It is likely that people from the database community or industrial sector would not even consider submitting a position article. A more complete comparison of specification methods would necessarily include these missing approaches.

One lesson I learned from the informal techniques is that injecting domain knowledge adds reality and complexity to a specificand. If such knowledge exists and if it can be added systematically, then incorporating a knowledge-based specification technique like Fickas's in the overall software-development process would be beneficial.

The formal specification techniques do not radically differ from one another. In fact, I was surprised by both the similarity among the state-transition models and the similarity among the logic-based models. (Perhaps I should not have been surprised, since all formal techniques are based on some common set of mathematical concepts.)

The popular, and generally accepted, technique for specifying an operation's effects is to use preconditions and postconditions. There is less agreement on how to specify data. Algebraic and set-theoretic approaches are common, but the dominant approach of the 12 papers presented is model-oriented, where you might model a set of books by a list of books and a book by a record of three components (title, author, and copy number). Such models are either overly restricted or implementation-biased.

I am confident that existing formal specification techniques can be used to identify many, but not all, deficiencies in a set of informally stated requirements, to handle simple and small problems, and to specify the functional behavior of sequential systems.

Except for perhaps the last, these are not new conclusions. In fact, they are reassuring and confirm claims made by many in the formal specification community. However, many challenges remain and are of interest to those active in software specification and design:

- demonstrating that existing techniques scale up or scaling up the techniques themselves,
- specifying nonfunctional behavior such as concurrency, reliability, performance, and human factors,
- combining different techniques such as a knowledge-based one with a more standard logic-based one, or an operation-oriented one with a data-oriented one,
- building tools, and
- integrating specification techniques with the entire software-development effort.

Finally, a reminder to the 12 papers' authors: It is the responsibility of an advocate of a particular specification method to tell potential users not only what the method is good for but also what it is *not* good for. Students (and readers) should not expect the method to be suitable for classes of applications and properties outside of the method's intended ones. However, students of a particular specification method should also not be expected to guess what those suitable classes are — teachers (and authors) must state the limitations of their methods. ❖

Acknowledgments

I thank Dick Kemmerer and Susan Gerhart for introducing the library problem, keeping it alive, and relaying its history to me; Mehdi Harandi for encouraging me to write this article based on the oral summary I gave at the fourth workshop on specifications and design; and the anonymous referees for their useful criticisms. Finally, I thank all the authors of the 12 papers for their timely response to my appeal for comments on and corrections to an early draft of this article.

This research was partly sponsored by IBM and the Defense Dept.'s Advanced Research Projects Agency under order 4864, amendment 20, contract F-33615-87-C-1499, monitored by the Avionics Laboratory at the Air Force Wright Aeronautical Laboratories. The National Science Foundation provided additional support under grant CCR-8620027.

References

1. R.G. Babb III et al., "Workshop on Models and Languages for Software Specification and Design," *Computer*, March 1985, pp. 103-108.
2. R.A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," *IEEE Trans. Software Eng.*, Jan. 1985, pp. 32-43.
3. D.R. Musser, "Abstract Data-Type Specification in the Affirm System," *IEEE Trans. Software Eng.*, Jan. 1980, pp. 24-32.
4. M. Feather, "Language Support for the Specification and Development of Composite Systems," *ACM Trans. Programming Languages and Systems*, April 1987, pp. 198-239.
5. B. Sufrin et al., "Notes for a Z Handbook. Part 1: The Mathematical Language," tech. report, Programming Research Group, Computing Lab, Oxford Univ., Oxford, England, Aug. 1984.
6. "Problem Set for the Fourth International Workshop on Software Specification and Design," *ACM Software-Engineering Notes*, April 1986, pp. 94-96.
7. D.L. Parnas, "The Use of Precise Specifications in the Development of Software," in *Information Processing 77*, North-Holland, Amsterdam, 1977, pp. 861-867.
8. J.M. Wing, "A Larch Specification of the Library Problem," *Proc. Fourth Int'l Workshop Software Specification and Design*, CS Press, Los Alamitos, Calif., 1987, pp. 34-41.
9. D.L. Parnas, "Software Aspects of Strategic Defense Systems," *American Scientist*, Sept.-Oct. 1985, pp. 432-440.
10. M.A. Jackson, *Principles of Program Design*, Academic Press, Orlando, Fla., 1975.

Jeannette M. Wing is an assistant professor of computer science at Carnegie Mellon University. Her research interests include formal specifications, programming languages, concurrent and fault-tolerant distributed systems, and visual languages. She contributed to the design of the Larch family of specification languages and now codirects the Avalon and Miró projects at Carnegie Mellon.

Wing received a BS, MS, and PhD in computer science from the Massachusetts Institute of Technology. She is a member of the ACM.

Address questions about this article to the author at Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA 15213-3890; CSnet jeannette.wing@cs.cmu.edu.

IEEE Software

SHEDDING LIGHT ON 4GL'S

HERE FOURTH-GENERATION LANGUAGES FIT IN

ALSO: OS/2 REVIEW • VISUAL UNIX • BRIDGING LOOPS AND PROLOG
APPLICATION GENERATORS • ICON-BASED PROLOG • HASHING SCHEME
SPECIFICATION EXERCISE • MILLS ON KEEPING IT SIMPLE

IEEE INSTITUTE OF ELECTRICAL AND