A Study of Twelve Specifications of the Library Problem

Jeannette M. Wing
26 July 1987
CMU-CS-87-142

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520. Additional support was provided in part by the National Science Foundation under grant DMC-8519254.

A Study of Twelve Specifications of the Library Problem

Jeannette M. Wing

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-3068
wing@k.cs.cmu.edu

26 July 1987

Abstract

Twelve workshop papers [25] include an informal or formal specification of Kemmerer's library problem [28]. The specifications range from being knowledge-based to logic-based to Prolog-based. Though the statement of the informal requirements is short and "simple," twelve different approaches led to twelve different specifications. All twelve, however, address many of the same ambiguities and incompletenesses, which we describe in detail, present in the library problem. We conclude that for a given set of informal requirements, injecting domain knowledge helps to add reality and complexity to it, and formal techniques help to identify its deficiencies and clarify its imprecisions.

A Study of Twelve Specifications of the Library Problem

Jeannette M. Wing

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

26 July 1987

1 Introduction

The purpose of this paper is to summarize and compare twelve different papers that address the same set of informal requirements—Kemmerer's library problem—as assigned to participants of the Fourth International Software Specification and Design Workshop held in Monterey, California in April 1987. Some of the specifications contained in the papers are formal; some, informal. Some authors followed a particular specification method or used a particular specification language. This paper focuses on what each of the methods or languages elucidates of the problem statement.

1.1 History of problem

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 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 meeting of this workshop (under a different title) that took place in Orlando, Florida [1]. In 1985 Kemmerer's *IEEE Transactions on Software Engineering* paper on testing formal specifications included an Ina Jo specification of the problem [12]. Finally, in 1986 in the Call for Papers [28], the organizers of the fourth workshop 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 proceedings of the workshop [25], twelve addressed the library problem. This paper discusses only those twelve, although other library specifications have been written, in particular Kemmerer's in Ina Jo, Gerhart's in Affirm [20] (unpublished), and King and Sorensen's in Z [31] (unpublished).

1.2 Informal Requirements

What follows is the statement of the library problem as it appears in the Call for Papers (K-Call) [28].

> Consider a small library database with the following transactions:
>
> 1. Checkout 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:
>
> 1. All copies in the library must be available for checkout or be checked out.
> 2. No copy of the book may be both available and checked out at the same time.

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

The above statement (K-Call) is slightly different from Kemmerer's original statement (K-TSE) in the following ways:

- K-TSE asks us to consider a "university library database."
- K-TSE does not restrict Transaction 1, checkout and return, to only staff users.
- K-TSE adds the following fourth constraint:

  4. A borrower may not have more than one copy of the same book checked out at one time.

Some of the authors of the twelve papers were familiar with Kemmerer's original statement, which naturally added to the variation among the interpretations of K-Call.

1.3 Guide to This Paper

The intention of most of the twelve papers is not to give a straightforward specification of the library problem, but to use it as a means of exemplifying an author's point of view about software specification. Authors of those papers use the library problem to make their points more concrete. Some papers use the library problem to illustrate a particular specification method or language, but even in some of those papers, the authors give only fragments of a larger specification. Given that some of the papers do not intend to give explicit specifications, let alone "complete" ones, we feel we cannot do a true comparison of specifications. We also do not want to compare papers since 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. Rather, we prefer that the reader read the individual papers for more information about the different authors' views on software specification and design.

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

We distinguish between the *specificand*, i.e., that which is being specified, and the *specification*, i.e., the text which describes the specificand [10]. Here, we take the specificand to be the concept of a library as conveyed by the informal requirements of K-Call. Section 2 summarizes the key ideas behind the twelve papers and compares them according to some general criteria (e.g., level of formality). Section 3 enumerates issues particular to the specificand, as clarified by the papers' authors or the specification itself. Section 4 discusses the comparison and Section 5 states some conclusions.

2 Summary of the Papers and Specifications

2.1 A Glimpse at the Papers

The table in Figure 1 summarizes the papers according to their level of formality, which phase(s) of the software lifecycle the authors address, the papers' key ideas, and the specific specification language used, if any.

Authors	*Formality*	*Lifecycle Phase*	*Key Idea*	*Language/ Project*
Kerth	informal	requirements-informal spec.	Structured Analysis and human interface	
Fickas	AI	requirements-formal spec.	usage scenarios	KATE
Rich, Waters, Reubenstein	AI	requirements-formal spec.	Requirements Apprentice	Programmer's Apprentice
Lubars	AI	spec.-design	design schema	
Dubois, van Lamsweerde	logic	requirements-formal spec.	meta-specification	
Wing	logic	requirements-formal spec.	benefits of formalism	Larch
Rudnicki	logic	analysis	testing and proving	Ina Jo
Yue	logic	analysis	sufficient-completeness and pertinence	GIST
Levy, Piganiol, Souquieres	logic and executable	requirements-formal spec.	multiple methods	SASCO
Terwilliger, Campbell	logic and executable	design-code	Anna and Prolog	PLEASE
Lee, Sluizer	executable	requirements-formal spec.	models of behavior	SXL
Rueher	executable	design-code	rapid prototyping and graphical Prolog	Prolog

Figure 1: Summary of the Papers

2.1.1 Formality

A specification is *formal* if it has a precise and unambiguous semantics. It is *informal* otherwise. A precise and unambiguous semantics is given by mathematics, usually in the form of a set of definitions, a set of logical formulae, 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 is a machine-executable interpreter, e.g., a Prolog interpreter, we consider the (formal) specification to be *executable.*[1] Usually a formal specification is 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, then a specification written in it is formal.

Informality is introduced by the reliance on English and/or uninterpreted diagrams in writing and giving the semantics of the specification. Of the four informal specifications presented in the papers, three use domain knowledge (indicated as "AI" in the table). All four are presented with tool support in mind so that some amount of machine processing, e.g., pattern-matching on keywords, could be performed on the

[1] "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 examples of nontrivial dimensions [1]." In this paper, we do not classify such specifications as executable.

specifications.

The underlying basis for all but two (Lee and Sluizer, Rueher) of the formal specifications is full first-order predicate logic. More specifically, pre- and postconditions are used to specify state transitions; other assertions, ranging from algebraic equations to first-order logical formulae, are used 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 (see Section 3.2.5).

Levy et al. use a Lisp interpreter to execute their specifications, performing only partial evaluation for incompletely defined functions. The three other executable specifications use Prolog. Terwilliger and Campbell start with a logic-based language, but proceed to use Prolog to "execute" pre- and postconditions, resulting in a prototype implementation written in Prolog. They explicitly acknowledge the loss of logical power because of Prolog's closed-world assumption. For example, there is no way to specify "the fact that UNDER_LIMIT is not true if the number of books checked out by the borrower is greater than the borrowing limit" [32]. Lee and Sluizer choose 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 [14]. Rueher uses Prolog as both an implementation and specification language. Thus, his logic is restricted to Horn clauses and is further "corrupted" by his indirect use of assert and retract predicates.

The virtues of formal specifications have been argued elsewhere [23, 2, 10], and with respect to the library problem in [33].

2.1.2 Lifecycle Phase

Most of the authors assume a traditional software lifecycle development process broken up into various phases: (1) stating *informal requirements*; (2) writing a *specification*, informal or formal or both; (3) developing a *design*; (4) writing the *code*; (5) *testing* and/or *verifying* the code. One paper focuses on elaborating informal requirements; henceforth, we consider "elaborated requirements" as an "informal specification." Six papers address the transition from informal requirements to formal specifications; however, two (Fickas, Rich et al.) stop short of giving formal specifications. One paper focuses on the transition from (informal) specifications to design; two, from design to code. Two papers address doing *analysis* of the specification itself, both during and after its development. The three papers that use Prolog advocate rapid prototyping and regard an executable specification as a prototype. Two papers (Dubois and van Lamsweerde, Levy et al.) address the specification process itself, and how to specify the actions, such as refinement and abstraction, performed in the process of specifying.

2.1.3 Key Idea

Below is a one-paragraph summary of each paper.

For specifying "real-life" systems, **Kerth** [13] notes the inadequacy of the *Structured Analysis* design method as described by DeMarco in [3], and augments it with a *Three Dimensional Human Interface Perspective* (3D-HIP). 3D-HIP consists of (1) a collection of graphical *views* that roughly look like displayable screen menus from which a user can select what function to do next; (2) a view transition diagram that represents the flow of control as a user moves from one view to another as different

functions are selected; and (3) a textual description of the operational behavior of a user interface, including what normal or undesirable events occur when keystrokes are entered or mouse buttons are clicked.

Fickas [7] describes the components of a system, KATE, used to automate the process of transforming informal requirements into a formal specification. The paper shows how to criticize refinements of informally stated requirements through the use of domain knowledge, *usage scenarios*, and *intermediate summaries*. Fickas also relies on human experts, e.g., professional librarians, to aid in critiquing a specification and to make KATE smarter by enlarging its knowledge database.

In the context of the ongoing Programmer's Apprentice research project, **Rich, Waters, and Reubenstein** [26] discuss using a *Requirements Apprentice* (RA) to assist a user in converting an initial informal requirement into a formal specification. The RA relies on simple deductive methods applied to extensive domain knowledge represented as *cliches*. For example, the library example uses cliches 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 [19] addresses design reusability by defining abstract graph representations of designs and then instantiating them to yield specific designs. In particular, he instantiates an inventory-control schema with domain knowledge (library databases) to model the library.

Dubois and van Lamsweerde [5] discuss the meta-issue of how to specify the process of specifying. Based on a dual object-model and operation-model of specification, they suggest two meta-models in which the process of specification is made explicit: the *process-model* captures the steps used by a specifier while constructing a specification; and the *method-model* captures the control information and rationale used for the steps taken, yielding an overall specification strategy.

Wing [33] demonstrates the benefits of formal specifications by identifying numerous problems with informal statements of requirements such as in the library example. She presents a specification of the library using Larch, which combines algebraic and predicative specification techniques into one framework.

Rudnicki [27] argues for detecting errors in specifications 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. His specification is based on Kemmerer's Ina Jo specification.

Yue [34] formally defines two properties, *sufficient completeness* and *pertinence*, both with respect to a set of *goals*. They capture the notion of whether a specification contains enough, but no more than necessary, information to achieve the goals. He discusses how to analyze a specification in terms of these properties, using a GIST specification of the library problem as an example.

Like Dubois and van Lamsweerde, **Levy, Piganiol, and Souquieres** [16] are interested in the specification process itself. They briefly describe SASCO, a system for supporting the evolution of an informal description into a formal specification. Through operations like refinement, enrichment, reuse, and abstraction, SASCO supports a multi-method approach to specification, where *method* means a

description of the process of specifying. Hence, instead of being limited to just one approach, e.g., top-down or data-oriented, SASCO supports several specification approaches.

Terwilliger and Campbell present PLEASE, an executable specification language that combines Anna ("annotated" Ada) and Prolog. PLEASE models data with Ada types, and specifies transactions with Ada procedures along with annotations that define pre-and postconditions and auxiliary predicates. Pre- and postconditions are written in terms of Horn clauses, and thus, are directly translatable into Prolog and executable by an interpreter.

Lee and Sluizer [15] assert that building and analyzing models as practiced in traditional engineering disciplines should be done in software engineering as well. They present SXL, an executable specification language that is based on a state-transition model. SXL uses transition rules via pre- and postconditions to specify allowed behavior, and logical invariants to specify required behavior. The invariants are automatically enforced during the execution of the model.

Rueher presents a graphical syntax for Prolog, which is used as an executable specification language. He endorses rapid prototyping through executable specifications, obtained by using Prolog as both his specification and implementation language.

2.1.4 Language, Project

The table in Figure 1 indicates the name of the specific language that the authors use to write the library specification, or the software project that they discuss in the papers. In each case, tools exist to perform automated syntactic and limited kinds of semantic analyses on the specifications.

2.2 A Glimpse at the Specifications

We distinguish between a *specification method* and the specification, i.e., that (piece of text) which results from following a method. The table in Figure 2 categorizes the specifications in the papers along three lines:

1. whether the specification is operation-oriented or data-oriented (also known as "object-oriented");
2. whether it is composed of separate modules;
3. whether graphics or diagrams are used to aid its readability.

The first two criteria are purely subjective.

2.2.1 Operation- versus Data-Orientation

Modern programming methods encourage the use of data abstraction as the primary organizational tool in program design [18], and correspondingly, modern programming languages like Ada [4], Smalltalk [9], CLU [17], and C++ [30], have explicit linguistic support for implementing data abstractions. Specification methods that focus on data can enjoy the same benefits that similar program design methods enjoy. Resulting specifications would additionally correspond more closely to an implementation written in a language supporting data abstraction.[2] The table indicates the three specifications that are not operation-oriented. Dubois and van Lamsweerde pay equal attention to both data and operations, whereas Wing

[2]As an aside, note that Fickas, and Rich et al. are concerned with just the opposite problem: not how to make the specification resemble the code, but how to make it resemble the informal requirements.

Authors	*Orientation*	*Modularity*	*Readability*
Kerth	operation		graphics
Fickas	operation		
Rich, Waters, Reubenstein	operation		
Lubars	operation		schemata
Dubois, van Lamsweerde	both	yes	
Wing	data	yes	statecharts
Rudnicki	operation		
Yue	operation		
Levy, Piganiol, Souquieres	operation	yes	
Terwilliger, Campbell	operation	yes	
Lee, Sluizer	operation		
Rueher	data		graphics

Figure 2: Summary of the Specifications

and Rueher focus on data. Dubois and van Lamsweerde, and Wing specify the semantics of data in terms of algebraic abstract data types. Since Rueher's ultimate concern is to transform a high-level design into code, he generates from his Prolog specification a high-level design consisting of Ada package definitions for types; data semantics are still in terms of Prolog predicates. The other specifications focus more on specifying the effects of the operations, and not the properties of the data. Note that this orientation could be attributed to the operational (transactional) presentation of K-Call, and not on the particular orientation of the authors' specification method.

In fact, the orientation of a specification *method* may be different from the orientation of a resulting specification. For example, contrary to what the table may imply, two other specification methods could be used in a way to generate specifications that are just data-oriented or both operation- and data-oriented. As previously mentioned, Levy et al. support a multi-method approach; the user is free to choose one or design his or her own. Also, although Terwilliger and Campbell's actual specification focuses only on the operations of interest, PLEASE'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 likely to be too simplistic. A dual specification method (Dubois and van Lamsweerde), or more generally, a multi-methods approach (Levy et al.), would be more appropriate. Giving a formal meaning to a specification that results from a mix of methods remains a challenge and is of current interest in the research community.

2.2.2 Modularity

For programs and large system design, the benefits of modularity, such as increased modifiability, reusability, and readability, are well-known [22]. For specifications, they are equally as important. Various modularity techniques, such as inheritance, type abstraction, and parameterization, 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, be used to 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 instead of using it as intended, and second, it may be hard to avoid spreading some design decisions throughout an entire specification, e.g., deciding that only one staff person can be hired to run a library [8].

The table indicates which of the formal specification approaches explicitly supports a modular construction of specifications. A blank entry for a formal specification indicates that a single specification is used to describe the behavior of the entire system. Not enough information was provided to determine whether or not any of the informal techniques has explicit support for modularity.

2.2.3 Graphics

Kerth uses graphics as a way to specify the human interface between a user and the library system. Lubars uses schematic diagrams (labeled circles and labeled arrows) to describe different kinds of database systems and their instances. Wing uses Harel's statechart notation [11] to illustrate the subset relations between different kinds of users and between different kinds of books. Rueher adds 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 is especially helpful in aiding in the understanding of his model of the library problem. He presents a collection of "views" of MacIntosh-like screen displays at different levels of the system hierarchy to simulate what a user would see on a terminal (or workstation), and a state transition diagram that depicts the flow from view to view as different transactions are performed.

3 Library Insights

The process of refining the informal requirements of the library example toward either a more detailed informal specification or a formal specification reveals different kinds of problems, such as inconsistencies, oversights, ambiguities, and incompletenesses, with informal description. Here, we consider two broad classes of problems with the informal statement of the library example: ambiguities and incompletenesses.

3.1 Ambiguities

The table in Figure 3 summarizes how each paper treats each of the five ambiguities we discuss in detail below. A blank entry for a paper that presents a formal specification indicates that insufficient information was given in the paper for us to determine one interpretation over another. Entries for informal specifications indicate what an author explicitly states in the paper. Whereas we could infer information from a formal specification and often did not rely on what the authors state explicitly in text, we chose not to try to infer from informal specifications.

Authors	*library*	*user*	*book*	*available*	*last checked out*
Kerth	yes	= 2	book ≠ copy		
Fickas		> 2		redundant	last = current
Rich, Waters, Reubenstein	yes		book ≠ copy		
Lubars	yes				last = current
Dubois, van Lamsweerde	yes	> 2	book ≠ copy	⇒ in library	
Wing		> 2	book ≠ copy	⇒ in library	last = current
Rudnicki		= 2	book ≠ copy	redundant	
Yue	yes		book ≠ copy		
Levy, Piganiol, Souquieres	yes	= 2	book ≠ copy	⇒ in library	
Terwilliger, Campbell		= 2	book = copy	⇒ in library	last = current
Lee, Sluizer	yes	= 2	book ≠ copy	⇒ in library	last ≠ current
Rueher	yes	= 2	book = copy	⇒ in library	last ≠ current

Figure 3: Ambiguities

3.1.1 What is a library?

The table indicates with a "yes" which authors explicitly distinguish between a library *database* and the entire library *system*. A library database includes records of books (e.g., author and title, and perhaps copy number) and records of users (e.g., 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," or "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 of the concept of the library is part of the specificand (the library) and what is part of the specificand's *environment* [10]. If the library is just the "database," then the environment must include the people who have access to the database, i.e., the people who perform the transactions on it. If the library is the entire "system," including the people (and books), then 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 the other public services (if a public town library). Though some of the authors discuss this ambiguity, none of the specifications makes clear the distinction between the specificand and its environment. In fact, the GIST specification language [6], used by Yue, models "closed-systems" which by definition both a system and its immediate environment.

3.1.2 What is a user?

Most specifications assume that library users are divided into two disjoint classes: those with the privileged library staff status and those without ("ordinary borrowers"). The table indicates with a "= 2" which specifications divide 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 (Dubois and van Lamsweerde) is that a user of the system is possibly different from a user of the database. In this case, a user of the library database is not necessarily the same as a person of library status since an ordinary borrower might use the database to find out what books he or she has checked out. Wing makes a distinction between a person affiliated with the library and someone who is not; only those affiliated with the library can have either staff or ordinary status. A third interpretation due to Fickas is based on the original statement (K-TSE) which qualifies the library be a "university" library database. Fickas reports that a professional librarian interpreted "user" to mean ordinary borrowers and "staff" to mean "organizational" staff, i.e., university staff as opposed to university faculty or students. In this case, university staff people would presumably be more privileged than university professors.

A consistent interpretation of both "library" and "user" is crucial in order to interpret the restriction that Transactions 1 (check out, return) be performed by staff users. A straightforward interpretation of the restriction is that only staff members are allowed to check out or return a book and that ordinary borrowers are not allowed to do either. This is more than an unreasonable situation—we have the impossible situation in which ordinary borrowers can find out what books they have checked out, but cannot check out books. A more reasonable interpretation is that only staff members can check out or return a book, and they do so *on behalf* of any ordinary borrower. Here, three objects are involved: the book, the staff member, and the borrower. A third possibility exists: a careful reading of Kemmerer's original statement (K-TSE) led Wing to believe that this restriction was an unintentional mistake in the Call for Papers (K-Call).

3.1.3 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 title (and perhaps, 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.

The table indicates whether the specification associates a unique identifier with the idea of a "book," thus equating the notions of "book" and "copy" (book = copy) or with the idea of a "copy," thus modeling a book as a set of copies (book $\neq$ copy). In two cases, the unique identifier is not explicitly modeled, but assumed to be associated with each physical entity in the system (Lee and Sluizer, Wing).

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 since if it had said "copies of books" it would then be consistent with other uses of "copy of book." If so, then 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 to distinguish between the numerous copies a borrower may have of the same book.

Another possibility exists (Wing): K-Call lacks Kemmerer's original fourth constraint which states that a user may have only one copy of a book checked out at once. This constraint is consistent with the informal statement of Transaction 4 in K-Call since now it would be clear that the transaction need not be concerned with returning copy numbers as well as book identities (author and title).

3.1.4 What does "available" mean?

Two papers (Rudnicki, Fickas) consider Constraints 1 and 2 as stating the same thing (indicated as "redundant" in the table). A book is either available or checked out; it cannot be both.

Other authors (see "$\Rightarrow$ in library" in the table), however, distinguish between not only whether a book is available or not, 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 do not cause one to be a restatement of the other.

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

3.1.5 What does "last checked out" mean?

Fickas's professional librarian notes that of the books that are on the shelves it is not interesting to find out who last borrowed them so "last" must mean "currently" (last = current). Three others authors 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. Equating the notions means that Transaction 5 returns a current borrower.

Lee and Sluizer, however, interpret "last checked out" to be different from "currently checked out" (last $\neq$ current) 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. 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 interprets "last checked out" to be different from "currently checked out." His Transaction 5, however, faithfully reflects the informal specification and returns the last borrower of only available books (and no current borrowers).

3.2 Incompletenesses

There are many kinds of incompletenesses in the informal requirements. The table in Figure 4 summarizes the six different incompletenesses we discuss in detail below. We will not discuss undefined terms like "title" or "subject," which could also be classified as a kind of incompleteness.

3.2.1 Initialization

As the table indicates, three papers explicitly characterize what properties must hold in the initial state of the system. In the state-transition model used by Lee and Sluizer and by Rudnicki, properties that must hold in the initial state are explicitly written in the specification. Lee and Sluizer specify that initially there exists a normal user, a staff user, an available book, and the book's entry in a card catalog (the library

Authors	*Initialization*	*missing operations*	*error handling*	*missing constraints*	*change of state*	*non-functional*
Kerth			error			human factors
Fickas			signals			system
Rich, Waters, Reubenstein		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	explicit no change	system
Yue			pre	yes		liveness
Levy, Piganiol, Souquieres		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

Figure 4: Incompletenesses

"database"). Rudnicki states 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 proves from his specification that to start any interesting activity in the library system, there must exist a user of library staff status. That person can then add a book to the library, which can then be checked out, later returned, etc.

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

3.2.2 Missing Operations

Some papers note the inadequacy of the given set of transactions, and propose adding some missing operations. For example, the following two operations are useful if a distinction between a book and a copy is made (see Section 3.1.3):

- Add a new book, as opposed to a copy of a book (Levy et al., Lee and Sluizer).
- Remove all copies, as opposed to a single copy, of a book (Rich et al.), and thus remove the existence of a book (a set of copies).

The following two operations are necessary in order to establish a state from which library activity can begin (see Section 3.2.1):

- Create a library (Wing).
- Add a staff user (Wing).

The following two operations are strictly not necessary. Without the first, however, there would be no need to distinguish between two types of users, if the operation of adding a staff user is included as above. Including the second makes the set of transactions more closely reflect reality, and more symmetric, if adding users are included.

- Add a regular user (Wing).
- Remove a user (Wing).

3.2.3 Error Handling

The informal requirements do not state what should happen if an error or undesired situation is encountered, e.g., trying to return a book that has not been checked out. A specification could either strengthen the precondition of a transaction in order to prevent the undesired situation from arising or it could strengthen the postcondition by explicitly specifying behaviors for the exceptional cases. In strengthening the postcondition, one could use a single "catchall" error or treat each exceptional case individually. The table indicates whether the specification handles errors either by strengthening only the precondition ("pre"), using a single catchall error ("error"), tuning error handling for different situations ("signals"), or some combination ("pre + X") of preconditions and error handling.

Some of the undesired situations that the authors address include:[3]

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

Finally, for completeness, specifications should treat type errors. If an argument or result of an operation is of the wrong type, then the specification contains an inconsistency. All of 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 pre- and postconditions of individual operations or defined in the underlying semantics by using predicate logic with typed variables.

3.2.4 Missing Constraints

In an informal sense, all authors added more "constraints" to K-Call, simply by (informally) elaborating the requirements, or making them unambiguous and more precise. The AI papers added domain-specific constraints, for instance by introducing knowledge about information retrieval systems for which a library is a special instance.

In a more formal sense, however, a "constraint " can be defined to be a state invariant to be maintained

---

[3]Recall that not all authors gave specifications of all transactions and for informal specifications, the authors may have only discussed the problem in text.

across state transitions in the execution of individual transactions. The table indicates of the formal specifications, which authors explicitly added more constraints to K-Call. 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 (Wing, Rudnicki).
- A borrower cannot keep a book indefinitely (Yue).
- There is a one-to-many relation between a book and its copies in the library (Dubois and van Lamsweerde, Lee and Sluizer).
- There is a one-to-one relation between a book and its "last" borrower (Lee and Sluizer).

For the latter two constraints, Lee and Sluizer additionally guarantee that in order to maintain the invariant, an object (e.g., a card-catalog entry) is automatically deleted when necessary (e.g., 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 pre- and post-conditions of each operation. Often such global invariants are discovered in the process of specifying an individual operation. For example, an implicit (unmodified) pre-condition 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, one can "distribute" an invariant to just the pertinent operations by affixing the appropriate pre-condition and/or post-condition. For example, K-Call's Constraint 3, which limits the number of books a borrower may have, shows 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 a pre- or post-condition of a particular operation needs to be strengthened.

3.2.5 Specifying Change of State

Some of the specifications are precise as to what objects change from state to state. In Ina Jo, as used by both Rudnicki and Kemmerer, the explicit NC"(x) assertion states that x's value does not change from the current state to the next. Lee and Sluizer state that "each postcondition includes only changes to the system state: values that are not explicitly mentioned are unchanged." In Larch, as used by Wing, a **modifies** $[x_1, ..., x_n]$ clause states that the only objects whose values are allowed to change are those in $\{x_1, ..., x_n\}$. Finally, Lubars mentions the notion of explicit "update records" which implies that all changes in state must be recorded explicitly. With a means of explicitly stating what is changed or implicitly stating what is not, the following kind of constraint can be made precise:

- The responsibility of a user for a book changes when the user checks out or returns the book. (Wing, Rudnicki, Lee and Sluizer)

3.2.6 Specifying Non-functional Behavior

Three kinds of non-functional behavior are explicitly addressed in the papers: *human interaction* with the library system, *system constraints*, and *liveness*. Only one paper (Kerth) addresses human factors by describing a menu-driven interface for depicting how people would interact with the system.

Two system constraints left unspecified are the borrowing limit of a user and the size of the library. Many authors introduce a uninterpreted variable like "max" or "limit" to denote the borrowing limit, but place no further constraints on it. Dubois and van Lamsweerde, however, constrain it to be non-negative. Rudnicki moreover constrains it to be positive, not simply non-negative, a restriction he discovered in the

process of testing his specification. Lee and Sluizer actually provide a limit (= 5) in their specification for the purposes of making their specification concrete and hence "fully" executable. Note that Constraint 3 says "books," not "copies," which reraises the question of "what is the difference between a book and a copy?"

Fickas discusses at length the implication of removing the borrowing limit constraint. For example, he notes that placing a borrowing limit may prevent a user who needs more books than allowed from achieving his goal. He also questions what "small" means. "Small library database" (K-Call) could mean a small-library database, a small library-database, a small-time system, or a simple problem involving a library database.

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

Yue explores the constraint on borrowers even further. He argues that progress could be impeded if either of the following problematic situations arises:

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

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

Dubois and van Lamsweerde do not discuss liveness explicitly but introduce enough formalism, in particular a sequence of times, so that they could characterize liveness properties. For instance, they use these times to determine whether a book has been returned by checking to see that each returned date associated with the book is less than the last checkout date.[4]

4 Discussion

We chose to compare the specifications according to how they address problems of the library example in order to illustrate the imprecision of natural language specifications and how twelve different approaches to the same set of informal requirements reveal many of the same problems. We admit that the revelation of a problem may be due to the authors' cleverness, and not to the particular approach they use; however, each of the approaches used undoubtedly helps to prod each of the authors into considering certain aspects of the informal requirements, and perhaps not others.

Our comparison highlights for the reader 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 of the twelve cases, 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 a systematic application of formal, or even informal, specification methods.

---

[4]This is an informal paraphrase of what they actually specify.

We chose only a subset of the ambiguities and incompletenesses present in the informal statement because those chosen represent those brought out most often by all twelve of the papers, and those not chosen would not have further revealed anything 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 (6-8 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, we are able to appeal to the reader's intuition about libraries.

On the other hand, its simplicity is deceptive in two ways: (1) Real libraries are not simple. They involve more than just people, books, and a (computerized) 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 different policies. In fact, Fickas evolves his library model from "talking to both library scientists and analysts, studying library science literature, and finally, looking at specific libraries" [7]. (2) 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 more complicated one. This second deception is compounded by the first, if we do not have the luxury of going to experts for their advice, of researching the literature, or examining existing systems [24].

Finally, none of the specifications mention the problems faced if the library operates in the presence of both concurrency and faults. The term "transaction" could be interpreted to imply that concurrent transactions be *serializable* [21], i.e., that they appear to execute sequentially. Transactions must always operate on a consistent state and leave the library in a consistent state upon their completion. For example, two checkout transactions can go on concurrently if they are checking out different books and are being performed by different staff members; however, they must be serialized with respect to each other if they are trying to check out the same book. Thus, serializability would be a desired property of a concurrent library system. "Transaction" could additionally imply the "all-or-nothing" property of database transactions. For example, if a borrower decides to abort in the middle of checking out a book, then the state of the library database should be as if the borrower never even started the checkout transaction. If observable states are at transaction boundaries, then temporary inconsistencies, e.g., a book is in the process of being checked out so is neither available nor checked out, would be hidden from the user. Thus, *failure atomicity* [29] would be a desired property of a fault-tolerant library system.

5 Conclusions

In reading the papers and specifications, comparing them, and comparing the various authors' points of view, we make the following observations.

One lesson learned from the informal techniques is that injecting domain knowledge adds reality and complexity to a specificand. If such knowledge exists and its addition can be done in a systematic way, then incorporating a knowledge-based specification technique in the overall software development process would be beneficial.

The formal specification techniques do not radically differ from one another. In fact, we were surprised by

both the similarity among the state-transition models and that among the logic-based ones.[5] The popular, and generally accepted, technique for specifying an operation's effects is to use pre- and postconditions. There is less of an agreement on how to specify data. Algebraic and set-theoretic approaches are common, but the dominating approach of the twelve presented is model-oriented where one might model a set of books by a list of books, and a book by a record of three components (title, author, copy number). We can confidently conclude 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 third, these are not new conclusions, but they are reassuring. Two broad challenges remain and currently are of interest to those active in formal specifications: (1) demonstrating that existing techniques scale up or scaling up the techniques themselves; and (2) specifying non-functional behavior such as concurrency, reliability, performance, and human factors.

Finally, we conclude with a reminder to the authors: It is the responsibility of each of the advocates of a particular specification method to tell potential users not only what the method is good for, but also what it is not. Each method is often intended for use on a specific class of applications, e.g., databases, and for specifying a specific class of properties, e.g., functionality. If so, the method should not be expected to be suitable for classes of applications and properties outside of their intended ones. However, students of a particular specification method should not be expected to guess what those suitable classes are; teachers 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. I also thank Mehdi Harandi for encouraging me to write this paper based on the oral summary I gave at the fourth workshop. Finally, I thank all the authors of the twelve papers for their timely response to my appeal for comments on and corrections to an early draft of this paper.

[5]Perhaps we should not have been surprised since all formal techniques are based on some common set of mathematical concepts.

References

[1] R.G. Babb II, R. Kieburtz, K. Orr, A. Mili, S. Gerhart, and N. Martin.
Workshop on Models and Languages for Software Specification and Design.
IEEE Computer 18(3):103-108, March, 1985.

[2] R. Balzer and N. Goldman.
Principles of good software specification and their implications for specification languages.
In *National Computer Conference*, pages 393-400. 1981.

[3] T. DeMarco.
Structured Analysis and Systems Specification.
Yourdon Press, 1979.

[4] Dept. of Defense.
Reference manual for the ADA programming language.
1983.
ANSI/MIL-STD-1815A-1983.

[5] E. Dubois and A. van Lamsweerde.
Making Specification Processes Explicit.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 169-177. Computer Society Press of the IEEE, April, 1987.

[6] M. Feather.
Language support for the specification and development of composite systems.
ACM Transactions on Programming Languages and Systems , 1987.
to appear.

[7] S. Fickas.
Automating the Analysis Process: An Example.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 58-67. Computer Society Press of the IEEE, April, 1987.

[8] S. Fickas.
private communication.
1987.

[9] A. Goldberg and D. Robson.
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley, 1983.

[10] J.V. Guttag, J.J. Horning, and J.M. Wing.
Some Notes on Putting Formal Specifications to Productive Use.
Science of Computer Programming 2(1):53-68, October, 1982.

[11] D. Harel.
Statecharts: A Visual Formalism for Complex Systems.
Science of Computer Programming 8, 1987.
to appear.

[12] R.A. Kemmerer.
Testing Formal Specifications to Detect Design Errors.
IEEE Transactions on Software Engineering SE-11(1):32-43, January, 1985.

[13] N.L. Kerth.
The Use of Multiple Specification Methodologies on a Single System.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 183-189. Computer Society Press of the IEEE, April, 1987.

[14] S. Lee and S. Sluizer.
private communication.
1987.

[15] S. Lee and S. Sluizer.
SXL: An Executable Specification Language.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 231-235. Computer Society Press of the IEEE, April, 1987.

[16] N. Levy, A. Piganiol and J. Souquieres.
Specifying With SACSO.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 236-241. Computer Society Press of the IEEE, April, 1987.

[17] B.H. Liskov, et al.
Lecture Notes in Computer Science. Volume 114: *CLU Reference Manual.*
Springer-Verlag, 1981.

[18] B.H. Liskov and J.V. Guttag.
Abstraction and Specification in Program Development.
The MIT Press, 1986.

[19] M.D. Lubars.
Schematic Techniques for High Level Support of Software Specification and Design.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 68-75. Computer Society Press of the IEEE, April, 1987.

[20] D.R. Musser.
Abstract Data Type Specification in the Affirm System.
IEEE Transactions on Software Engineering 6(1):24-32, January, 1980.

[21] C.H. Papadimitriou.
The serializability of concurrent database updates.
Journal of the ACM 26(4):631-653, October, 1979.

[22] D.L. Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM 15(12):1053-1058, December, 1972.

[23] D.L. Parnas.
The Use of Precise Specifications in the Development of Software.
In *Information Processing 77*, pages 861-867. IFIP, North-Holland, 1977.

[24] D.L. Parnas.
Software Aspects of Strategic Defense Systems.
American Scientist :432-440, September-October, 1985.

[25] IEEE Computer Society.
Proceedings of the 4th International Workshop on Software Specification and Design, IEEE Computer Society Press, 1987.

[26] C. Rich, R.C. Waters, and H.B. Reubenstein.
Toward a Requirements Apprentice.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 79-86. Computer Society Press of the IEEE, April, 1987.

[27] P. Rudnicki.
What Should Be Proved And Tested Symbolically In Formal Specifications?
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 190-195. Computer Society Press of the IEEE, April, 1987.

[28] Call for Papers.
Problem Set for the 4th International Workshop on Software Specification and Design.
ACM Software Engineering Notes , April, 1986.

[29] A.Z. Spector, J. Butcher, D.S. Daniels, D.J. Duchamp, J.L. Eppinger, C.E. Fineman, A. Heddaya, and P.M. Schwarz.
Support for Distributed Transactions in the TABS Prototype.
IEEE Transactions on Software Engineering 11(6):520-530, June, 1985.

[30] B. Stroustrup.
The C++ programming language.
Addison-Wesley, 1986.

[31] B. Sufrin, C. Morgan, I. Sorensen, and I. Hayes.
Notes for a Z Handbook: Part I--The Mathematical Language.
Technical Report, Programming Research Group, Oxford University Computing Laboratory, August, 1984.

[32] R.B. Terwilliger and R.H. Campbell.
PLEASE: A Language for Incremental Software Development.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 249-256. Computer Society Press of the IEEE, April, 1987.

[33] J.M. Wing.
A Larch Specification of the Library Problem.
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 34-41. Computer Society Press of the IEEE, April, 1987.

[34] K. Yue.
What Does It Mean To Say That A Specification Is Complete?
In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 42-49. Computer Society Press of the IEEE, April, 1987.