1

136 135

N93-17[illegible]8

On the Nature of Bias and Defects in the Software Specification Process

Pablo A. Straub
Computer Science Department
Catholic University of Chile

Marvin V. Zelkowitz
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

Implementation bias in a specification is an arbitrary constraint in the solution space. This paper describes the problem of bias and then presents a model of the specification and design processes describing individual subprocesses in terms of precision/detail diagrams, and a model of bias in multi-attribute software specifications. While studying how bias is introduced into a specification we realized that software defects and bias are dual problems of a single phenomenon. This has been used to explain the large proportion of faults found during the coding phase at the Software Engineering Laboratory at NASA Goddard Space Flight Center.

1 Introduction

Most informal software specifications are ambiguous, imprecise, and incomplete. Moreover, this is usually not evident by looking at a particular specification. This has prompted research on desirable and undesirable characteristics of specifications and specification languages. To make specifications precise, formal languages are used. Some of these languages are defined so that automatic compilation or execution is possible. However, much detail has to be included in executable specifications [5]. This extra detail not only makes the specification harder to read [6], but also leads to 'implementation bias'.

Alas, implementation bias—an arbitrary constraint in the solution space—is a term often used but not well defined. This has resulted in two effects: Either (1) specifications are biased, or (2) they are incomplete, for fear of bias. In fact, what has been called 'bias' in the literature is sometimes the desirable record of design constraints and design decisions. The problem of bias is related to the more important problem of software defects, because both are manifestations of either misconceptions with respect to the problem or preconceptions with respect to the solution; hence, we study these two problems together.

Overview of the Paper. This paper presents a model to help understand bias in software specifications. The remaining of this introduction presents our framework, the problem of bias and the concept of specification correctness. The next section presents our view of the process of specification and design. Section 3 presents our model of bias which is based both on the specification process and on a classification of requirements. Within this model, bias is not an absolute property of a specification, but depends on the process of creation of the specified requirements, that is bias depends on the process of specification and design. Section 4 presents the relationships that exist between bias and defects in a specification, and a study made at the Software Engineering Laboratory that explains the high relative incidence of coding faults in that environment.

1.1 Specification Framework

In this work we are considering multi-attribute specifications developed by starting from a description of requirements, and then refining it in several stages [3, Chapter 1]. Each stage takes a specification and produces a product, which is a more refined specification, until a program (i.e., a specification for a computation) is obtained. This view is not an endorsement of any particular development method: it models top down development, the waterfall life-cycle model, Boehm's spiral model, transformational programming, and other development methods.

We first define some related concepts.

Attribute: feature or dimension that characterizes software systems (e.g., average response time).

Requirement: constraint in the values of attributes (e.g., average response time shall be 0.5 seconds).

Preference measure: a measure of the goodness of the different values for a given attribute (e.g., smaller response time values are better).

Specification: statement of attributes, requirements, and preference measures for a software system.

Specificand set: set of all systems that satisfy a specification.

Solution set: set of all systems that solve a problem.

PRECEDING PAGE BLANK NOT FILMED

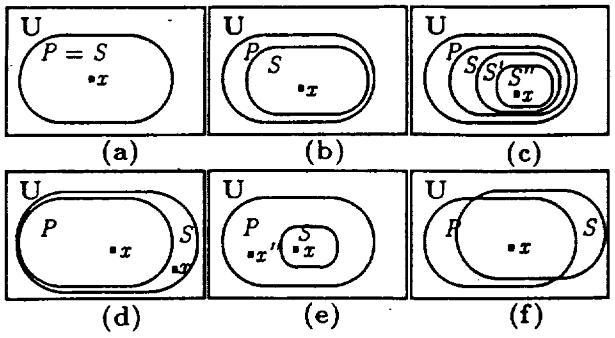

Figure 1: Specificand S, solution P, and particular solutions x and x'': (a) ideal, (b) acceptable initial specification, (c) successive specification stages, (d) incomplete specification, (e) bias, (f) usual case.

Whereas the specificand set is defined in terms of a particular precise specification of a problem, the solution set is defined in terms of the problem itself without reference to any written specification. That is, the specificand set comprises all systems that are correct with respect to the written specification, and the solution set comprises all systems that satisfy the user or customer. The differences between these sets are at the heart of our model; they are also the cause of defects in specifications.

1.2 The problem of bias

An ideal initial specification is general and precise enough so that a software system satisfies the specification if and only if it solves the problem at hand, that is, the specificand set equals the solution set (Figure 1a). This view is too optimistic, because there can be many solutions that do not even involve software. In practice, we only require software systems satisfying the specification to be solutions, and that no substantial class of solutions does not satisfy the specification, so that we can arrive at an optimal or nearly optimal solution (Figure 1b). An idealized development by staged specifications constrain the specificand set (Figure 1c) by adding design decisions—and nothing else. Incomplete specifications (Figure 1d) may lead to defects; for instance, x' satisfies the specification but it does not solve the problem. On the other hand, bias (Figure 1e) may lead to inefficiencies (e.g., optimal solution is really x'') and other development problems because the developers are overconstrained. Unfortunately, most specifications suffer both problems (Figure 1f).

A specification is biased if some of its requirements are arbitrary. Biased specifications overly constrain the specificand set, precluding some valid implementations as solutions to the problem at hand. Hence, the amount of bias is a common yardstick to judge software specification methods: those that are considered biased are usually rejected. Unfortunately, bias is sometimes confused with intended constraints in the solution set.

1.3 Avoiding bias

A generally accepted rule to avoid bias is "A specification should describe only *what* is required of the system and not *how* it is achieved."[1] However, this rule does not solve the problem: it only shifts it, because whether some requirement is a *what* or a *how* depends on one's point of view. For instance, the same requirement can be seen as a *how* by the designer and as a *what* by the implementor. During the process of refining the specification, some *how*'s become *what*'s: a design decision (i.e., *how* to do something) made by a designer is a requirement (i.e., *what* to do) for the implementor. A *how* becomes a *what* when a decision is made: a new requirement is incorporated into the current specification stage.

Consider a specification for a subprogram. The external interface of the subprogram is considered a requirement by the programmer (it is a *what*), because he or she cannot change it. This same interface was previously a *how* for the designer of the whole program, because he or she could have chosen an alternative interface. On the other hand, internals of the subprogram (e.g., algorithms, data structures, local variable names) are mostly *how*'s for the programmer, because he or she can change them.

There is no reason to include a *how* in a specification: specifications should describe what is desired and no more. However, often some attribute that is already fixed (i.e., it is a *what*) is not specified because of fear of bias. For instance, if within an institution there is a convention for local variable names for the purpose of easing maintenance, then the adherence to this convention is a *what*: It is already fixed, the programmer cannot change it, so it should be specified. We argue that this kind of constraint is not bias; in Section 3.3 we provide a definition of bias that is consistent with this view.

1.4 Specification Correctness

Specification bias and specification defects are intimately related. As can be seen from Figure 1, bias is related to the set difference of the solution set and the specificand set, $P - S$. That is, there is bias only if there are acceptable and preferred solutions outside the specificand set. Conversely, defects are related to the specificand set minus the solution set, $S - P$. That is, if an implementation i is unacceptable but is correct with respect to the specification, it is in the set difference (i.e., $i \notin P \wedge i \in S \Rightarrow i \in S - P$). In other words, bias and

[1] A common statement of this rule is "A specification should describe only *what* the system should do, not *how* it should do it." This modified rule is only useful with functional specifications: it views a software system as a specification for a computation, rather than as a product.

defects in the specification are dual problems.

Assume that for a given specification, the specificand set is contained in the solution set. In this case, all correct implementations are acceptable. This motivates the notion of specification correctness with respect to a problem, which is similar to the more familiar notion of implementation correctness with respect to a specification. (The main difference between these two concepts is that specification correctness cannot be formally verified because it is defined relative to an abstract problem.) A specification is correct if it is realizable (there is a correct implementation) and complete (all correct implementations solve the problem). That is, for a correct specification it is possible to derive an implementation and any implementation derived solves the problem. On the other hand, a specification is called impertinent to the problem if there is not a correct implementation that solves the problem.

The above is formalized as follows: Let S be the specificand set of a specification and let P be the solution set of a problem.

- The specification is *realizable* iff $S \neq \emptyset$.
- The specification is *complete w.r.t. the problem* iff $S \subseteq P$.
- The specification is *correct w.r.t. the problem* iff it is realizable and complete.
- The specification is *pertinent to the problem* iff $S \cap P \neq \emptyset$.

The following relations between these concepts are immediate: correctness implies pertinence ($S \neq \emptyset \wedge S \subseteq P \Rightarrow S \cap P \neq \emptyset$); pertinence implies realizability ($S \cap P \neq \emptyset \Rightarrow S \neq \emptyset$); completeness and pertinence imply correctness (because pertinence implies realizability); unrealizability implies completeness and impertinence ($S = \emptyset \Rightarrow S \subseteq P \wedge S \cap P = \emptyset$); there is no correct specification for a problem without a solution ($P = \emptyset \Rightarrow \nexists S\colon S \neq \emptyset \wedge S \subseteq P$).

To analyze the correctness of a specification with respect to a problem, consider the emptyness of the set $S - P$, related to the completeness of the specification, and of the set $S \cap P$, related to the pertinence of the specification. There are four cases: (a) The specification is unrealizable; (b) the specification is correct; (c) the specification is realizable but not pertinent; and (d) the specification is pertinent but incomplete, that is the specification can be made correct by adding more requirements. Figure 2 presents these cases, with case (d) comprising two subcases. In cases (a) and (c), the only choice is to backtrack, since at this point it is impossible to derive an acceptable solution. In case (b) there are no problems of correctness, but there can be problems of specification bias, if the preferred solution lies outside the specificand set as in Figure 1e. In case (d), the specification is incomplete, so addition of

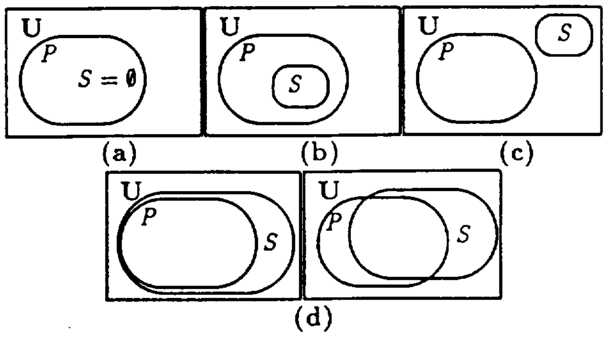

Figure 2: Specificand set S with respect to solution set P: (a) unrealizable, (b) correct, (c) realizable but impertinent, (d) pertinent but incomplete.

problem-specific information is needed to achieve a correct specification.

2 Specification Refinement

The specification and design processes are complex processes in which technical knowledge, art and inspiration take part [10]. Goel and Pirolli [4] describe the traditional view of design as a four-step process: "(1) an exploration and decomposition of the problem (that is, analysis); (2) an identification of the interconections; (3) the solution of the subproblems in isolation; and (4) the combination of the partial solutions taking into account the interconnections (that is, synthesis)."

In this work we go beyond these general processes and describe the subprocesses that occur specifically in software design. We characterize these subprocesses by how a current specification is updated to produce the next specification within a series, and also by how precision and detail are added to the specification. There is no assumption that all requirement analysis is done before design; on the contrary, requirements gathering and design are supposed to be intertwined [12].

2.1 Refinement Subprocesses

We assume that there is a written initial specification and that successive specifications will be created by a series of modifications to that specification. With respect to the subprocesses that perform these modifications—typically additions to the current specification—we postulate that there are four main kinds of activities that modify a specification:

Explication: addition of a requirement by making explicit a nonexplicit requirement.

Design decision: addition of a requirement by choosing a particular design.

Presentation change: change in the notation, presentation, or structure of the specification.

Retraction: withdrawal of a requirement from a previous decision or explication.

Even though we present these as discrete changes, actual changes to a specification usually involve a combination of them. For example, after finding an incorrect explication an analyst may replace the corresponding requirement by another one: a retraction followed by an explication.

Explication

Explication is one of the main activities during requirements gathering. Explications make the specification more complete, that is, ensure that software systems satisfying the specifications are solutions. In Figure 1 the goal is to transform a specification like (d) into one like (a). This goal is achieved by making explicit either *domain information*, *problem-specific information*, or *consequences of the specification*, thus reducing the specificand set.

Of course, the new requirement is not always a valid explication (e.g., something believed to be a consequence of the requirements might not be). This is intimately related to the concepts of specification correctness (Section 1.4) and bias (Section 3.3).

Design Decisions

As the name suggests, design decision is the most important process during design activities. Design decisions guide the implementation process towards a preferred set of solutions reducing the specificand set (as in Figure 1c). The information needed to make design decisions comes mainly from the previous specification and the solution domain. For example, semantic-preserving transformations in transformational programming are design decisions, because they preserve the functionality while improving other attributes of the algorithm.

We have identified several kinds of design decisions: decomposition, refinement, composition, abstraction, instantiation, reuse, creation of alternatives, and choice. Some of these are intimately related so we discuss them together.

Decomposition and refinement. Decomposition consists of dividing the problem into subproblems. It is usually followed by refinement, which means defining unspecified concepts or objects. These two processes are the core of stepwise refinement.

Composition. On the other hand, composition is the process of creating a solution to a problem by combining solutions to subproblems. That is, composition is the main process in bottom-up development. Composition is used most effectively in combination with reuse.

Abstraction, instantiation, and reuse. Abstraction as a design decision consists of specifying a solution to a more general problem (i.e., a problem of which the problem of interest is an instance), usually defining a set of (formal) parameters to describe particular instances. The rationale for solving more general problems is that it is often easier to abstract away particulars of the problem of interest and solve a general problem. Furthermore, the more general solution can be reused in other contexts.

Reuse as a design decision consists of prescribing the use of a particular solution to a subproblem. If the solution to be reused is parameterized (i.e., it has formal parameters) actual parameters must be provided to do the reuse. Instantiation is the process of defining actual parameters for a parameterized abstract solution.

A solution to reuse need not be already implemented: it may be simply specified as the solution to another subproblem. When several subproblems in the current design are instances of a single general problem, abstraction, instantiation and reuse can be employed to "factor" the design.

Creation of alternatives and choice. When it is not immediate which kind of design is the best, it is possible to create several alternative designs using some of these techniques. A valid implementation must conform to one of the created designs. After more elaboration of these designs, some are discarded until one design prevails. Choice is the process of selecting among alternative designs; the choice process is more objective when it is based on preference measures [2].

Presentation Changes

Presentation changes are intended to change the precision, formality, readability, modularity or other aspects of the specification itself, without affecting the specificand set, that is, without adding more information. For example, a condition written in English, referring to a collection of objects can be replaced by a logical predicate in which the collection is represented by a set.

Ideally, a presentation change does not change the specificand set, that is, it does not create new requirements. However, restrictions in the specification languages or methods used may impose additional constraints. In the above example, should our specification language support lists but not sets, we might have specified a list as an implementation for a set. If we later coded this list in Pascal we might have coded our list specification into an array or linked structure rather than the more efficient set data type that actually was originally specified. That is, as a result of a specification language deficiency we have added an additional arbitrary constraint for the program that resulted in it being less efficient, that is, we have added bias.

Retraction

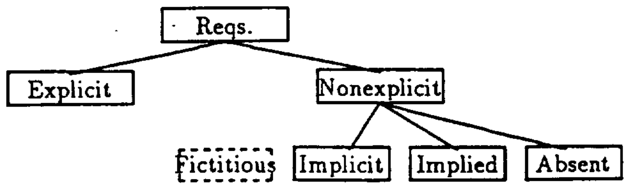

Figure 3: Classification of requirements: explicitness. Fictitious requirements are shown with segmented line because they are not real requirements.

Retraction occurs when a designer realizes that the current design is incorrect or otherwise undesirable. The goal of retractions is to create a *pertinent* specification, as defined in Section 1.4. As we said before, the retraction process is usually done in conjunction with other processes that create a new "replacement" requirement.

3 A Model of Bias

Presence of bias cannot be determined from the requirements alone, because it depends on the origins of requirements. For instance, if the origin of a particular requirement is in the problem, the requirement is not bias; if the origin is a misconception it may be. Hence, our definition of bias is based on a classification of requirements.

Requirements are classified into several classes with subtle differences. These subtleties are what makes bias hard to define and even harder to find. The main classification criteria we consider are explicitness and origin, which depends on the process of creation of new requirements.

3.1 Explicitness

A requirement is *explicit* if it is present in the specification; otherwise, it is *nonexplicit.*

Nonexplicit requirements are a recurring cause for misunderstandings in product development. They are further classified as follows (Figure 3).

Implicit requirements are those that are understood to be part of every product in the application domain, and so they are left unstated.

Implied requirements are logical consequences of other requirements.

Absent requirements are requirements unintentionally omitted in the specification, but are required by the solution set. These are not part of every product in the application domain.

Fictitious requirements [8] are assumptions made by the reader of the specification and not requirements at all: the reader believes that they are either implicit, implied or absent requirements.

A *real* nonexplicit requirement is either an implicit, implied, or absent requirement.

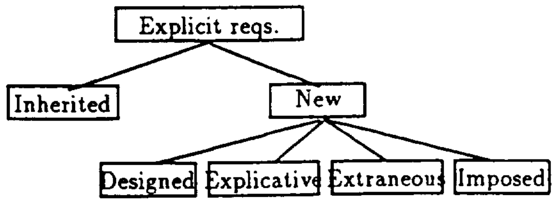

Figure 4: Classification of explicit requirements: origin.

3.2 Origin

An explicit requirement is *new* with respect to a certain specification stage if it is first made explicit at that stage; otherwise, the requirement is *inherited* from previous stages. (When the specification stage is clear from context we will say simply 'new' or 'inherited' requirement.) Of course, every explicit requirement is new to one stage, namely the stage in which it is introduced.

The discussion in Section 2 motivates the following classification of new requirements with respect to their origin (Figure 4).

Designed requirements are the consequence of design decisions taken at the current specification stage.

Explicative requirements are created by explication of implicit, implied, or absent requirements.

Extraneous requirements are created by explication of fictitious requirements.

Imposed requirements are those imposed by the limitations of the specification method or language used, created as a side effect of a presentation change.

This classification describes possible origins for the requirements, but it does not provide a method to determine the origin. For example, without a complete analysis of the application domain, there is no definite method to tell whether a requirement is extraneous or the explication of an implicit requirement.

3.3 The Nature of Bias

We define bias in terms of the origin of the requirements described in a specification: A specification containing extraneous or imposed requirements is *biased.*

This definition provides insight into the problem of bias, including both its origins and consequences. The origin of bias is either wrongful interpretation of nonexplicit requirements or the limitations imposed by the specification method. The consequences are that the specificand set can be overly constrained or that the solution adopted can be suboptimal. That is, a biased specification will lead the design towards particular implementations that are not necessarily the best possible.

The definition does not provide a method to measure bias content in a specification, because bias is defined in terms of the origin of requirements and we cannot be completely sure of the origin of some requirements. Furthermore, bias is relative to the application domain and the software engineering environment, because the domain and environment define what is implicit.

For example, in an environment in which all programs are written in a particular programming language, the presence of idioms of this language in a specification is not necessarily bias, unless another implementation language is introduced to the environment. This is what happened at the Software Engineering Laboratory (SEL) at the National Aeronautics and Space Administration (NASA).[2] During the first experience with development in the Ada language they realized that software specifications for satellite dynamics simulators were "heavily biased toward FORTRAN. In fact the high level design for the simulators is actually in the specifications document" [1]. This was not a problem—on the contrary, it facilitated both development and reuse of specification and code—until the first development in Ada: the specifications had to be rewritten first. Given our definition of bias these FORTRAN-oriented specifications were not necessarily biased; they contained many designed requirements. Before Ada was introduced, the use of FORTRAN was an *implicit* requirement. After that, the choice of appropriate language became an *explicit* attribute, resulting in the assumption of FORTRAN as a *fictitious* requirement.

The relative nature of bias is an essential characteristic. It stems from the existence of nonexplicit requirements and the inherent uncertainty with respect to those requirements. That does not imply that there is nothing to do: an obvious task is to make explicit as much as possible about the domain and environment. If this is done, we are reducing considerably the possibilities of bias. However, as long as there are nonexplicit requirements, there will be doubt about these requirements and hence possibility of bias. Making explicit the implicit requirements of a certain domain and environment still leaves two sources of bias: restrictions on the method and languages, and absent requirements. These two cannot be avoided completely: the first because any method that provides some guidance in the specification process will guide the design to some particular kind of solutions; the second because at the beginning of a project most requirements are absent.

[2]The SEL was created in 1976 to study and improve the software process at NASA Goddard Space Flight Center.

4 Software Defects

Both bias and software defects are a consequence of problems in the development process. Section 1.4 shows the duality of bias and faults by analysing the differences of the specificand set and the solutions set. Here this comparison is extended further. We classify software defects in three classes [11]: *faults* occur in documents, *errors* occur in human processes, and *failures* occur in automatic processes. There is an analogy between the problem of bias and defects: fictitious requirements are like errors (both during human processes), imposed and extraneous requirements like minor faults (both occur in documents), and inefficiencies like minor failures (both occur during automatic processes). The criticality of the attributes involved is related to whether something is considered a fault or simply bias.

During software development, successive specifications are written, usually starting from an incomplete specification towards a correct specification. Every specification inherits from all previous specifications, so if there is a new requirement that contradicts an explicit previous requirement the new specification is inconsistent and hence unrealizable. The only solution is to retract either the new requirement or previous requirements. Similarly, if a new requirement contradicts a nonexplicit real requirement the specification is made impertinent to the problem (i.e., it solves another problem); again, the only solution is to retract. All too often a specification is unrealizable or impertinent but this is not evident to the developers so no retraction occurs and development continues. This is a secondary but important source of defects.

We have studied these problems at the SEL. The software analyzed are ground support systems for unmanned spacecraft. Most systems are about 100K source lines FORTRAN programs, but a sizable percentage are now in Ada. The SEL has a database describing systems and their development processes made in the last 15 years. The analysis that follows uses data from that database, but only considers relatively recent data (since January 1, 1986), because the software process has changed.

Table 1 summarizes counts of change reports classified by type of change (e.g., requirement changes, fault correction) in all SEL projects. From the table, 49.4% of the changes are due to faults, 12.3% correspond to planned enhancements and 10.6% are due to requirements changes.

Table 2 summarizes counts of the changes due to the 8074 faults of Table 1, classified by source of fault. From the table, 74.8% of faults are related to coding and 16.3%

Type of change	Count	%
Fault correction	8074	49.4
Environment change	533	3.3
Improvement of user services	1205	7.4
Planned enhancement	2018	12.3
Presentation changes	1464	9.0
Requirement changes	1730	10.6
Other	1327	8.1
Total	16351	100.1

Table 1: Changes by type in SEL projects since 1986.

Fault source	All faults	
	Count	%
Requirements	76	0.9
Functional specification	242	3.0
Design	996	12.3
Subtotal specifications	1314	16.3
Code	6043	74.8
Previous change	714	8.8
Other	3	0.0
Total	8074	99.9

Table 2: Fault source in SEL projects since 1986.

of the detected faults are directly related to incorrect specifications (our definition of 'specification' includes three SEL phases: requirements, functional specifications, and design). This simple analysis demonstrates that up to 16% of all problems can be related to implementation bias in the specifications.

However, because requirements documents and their changes originate outside the SEL and within some requirements generation group at NASA, these changes are not considered faults in the specifications. If we assume that the 1730 requirements changes in Table 1 were indeed fault corrections, the total number of faults would be $8074 + 1730 = 9804$, the total number of specification faults would be $1314 + 1730 = 3044$ and hence specification errors would account for up to 31.0% of all faults. This assumption is not as extreme as it looks, because predicted changes in the requirements, improvements and environment (hardware) changes are classified separately. In summary, considering all faults, between 1/6 and 1/3 of all faults at the SEL are related to specifications, and potentially are related to implementation bias.

Another source of faults related to specifications are faults of omission: when something is not specified it is not a problem of the code but of the specification. The fact that the problem shows up during coding or testing does not mean that the problem is coding. Table 3 shows counts of faults of omission, commission, omission/commission separated by fault source (the 'Total' column is not identical to the 'All faults, Count' column from Table 2 because

Source	Comm.	Om.	Both	None	Total
Reqs.	19	40	8	9	76
Specs.	102	78	40	20	240
Design	253	550	159	34	996
Code	2302	2334	921	482	6039
Prev. chg.	289	295	79	50	713
Total	2965	3297	1207	595	8064
Percent	36.8	40.9	15.0	7.4	100.0

Table 3: Omission and commission faults in SEL projects.

10 faults had invalid data). At the SEL 37% of all faults are faults of commission, 41% are faults of omission and 15% are faults of omission/commission. Thus, about one half of the faults are of omission and potentially can be attributed to incompleteness in the specifications.

In conclusion, even though coding appears to be by far the most important source of faults, a deeper analysis of the specification process reveals that many coding faults have roots in earlier stages. Implementation bias undoubtedly plays an important role in many of these 3000 faults that are related to changes due to specification issues.

5 Conclusion

Even though bias is widely recognized as an undesirable property of specifications, it has not been adequately studied. This has caused confusion with the related concept of design decision, so that the presence of designed requirements in specifications has been considered undesirable. This is in contrast with the use of specifications in other engineering disciplines, where a specification may include many designed requirements (e.g., materials, manufacturing methods).

In this paper we presented a model to describe the nature of bias and distinguish bias from designed requirements and other requirements in a specification. This model is based on a classification of all the requirements described in a specification and also those that are not described (i.e., nonexplicit); it explains the nature of bias, but since it uses nonexplicit requirements it does not lead to any definite method to detect bias. However, the model does explain both the relative and unavoidable nature of bias. Because bias depends on the specification process we had to model that process. This modeling shed light on the problem of software defects, a relationship that in turn helped us to potentially explain the high relative number of coding faults found at the SEL.

Although we have developed an explanatory model of the design process, quantification of these concepts is needed before we can develop practical procedures for applying them in large scale developments. Additional work in this direction in continuing.

Acknowledgements

This research was supported in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland. Thanks to Sergio Cárdenas-García and Eduardo Ostertag for their helpful comments. P. Straub was partially supported by a scholarship from the Catholic University of Chile while he was at the University of Maryland.

References

[1] Carolyn E. Brophy, W.W. Agresti, and Victor R. Basili. Lessons learned in use of Ada-oriented design methods. In *Proceedings of the Joint Ada Conference*, March 1987.

[2] Sergio Cárdenas and Marvin V. Zelkowitz. Evaluation criteria for functional specifications. In *Proceedings 12th Int'l Conf. on Software Engineering*, pages 26–33, Nice, France, March 1990.

[3] Bernard Cohen, William T. Harwood, and Melvyn I. Jackson. *The Specification of Complex Systems*. Addison-Wesley, Reading, Massachusetts, 1986.

[4] Vinod Goel and Peter Pirolli. Motivation the notion of generic design within information-processing theory: The design problem space. *AI Magazine*, 10(1), spring 1989.

[5] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

[6] C.A.R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, pages 85–91, September 1987.

[7] Cliff B. Jones. Systematic program development. In N. Gehani and A.D. McGettrick, editors, *Software Specification Techniques*. Addison Wesley, Reading, Massachusetts, 1986.

[8] Edward V. Krick. *An Introduction to Engineering and Engineering Design*. John Wiley and Sons, New York, N.Y., second edition, 1969.

[9] Harlan D. Mills, Michael Dyer, and Richard C. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–24, September 1987.

[10] Ellen Shoshkes. *The Design Process*. Whitney Library of Design, New York, 1989.

[11] Pablo A. Straub and Eduardo J. Ostertag. EDF: A formalism for describing and reusing software experience. In *International Symposium on Software Reliability Engineering*, pages 106–113, Austin, Texas, May 17–18 1991.

[12] William Swartout and Robert Balzer. On the inevitable intertwinning of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.