Bias and Design in Software Specifications*

Pablo A. Straub†
Computer Science Department

Marvin V. Zelkowitz
Computer Science Department and
Institute for Advanced Computer Studies

University of Maryland
College Park, Maryland 20742

December 19, 1990

Abstract

Implementation bias in a specification is an arbitrary constraint in the solution space. While bias is a recognized problem, it has not been studied in its own right. 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.

This paper presents a model of bias in software specifications. Bias is defined in terms of the specification process and a classification of the attributes of the software product. Our definition of bias provides insight into both the origin and the consequences of bias: it also shows that bias is relative and essentially unavoidable. Finally we describe current work on defining a measure of bias, formalizing our model, and relating bias to software defects.

Keywords. Implementation bias, software design, software defects, requirements, formal specifications.

1 Introduction

Most informal software specifications are ambiguous, imprecise, unclear, incomplete, etc. Moreover, this is usually not evident by looking at a particular specification.

*This research is supported in part by NASA Goddard Space Flight Center, grant NSG-5123.
†Additional support from ODEPLAN Chile and the Catholic University of Chile.

The need to produce better specifications has prompted research in several features of specifications. Guttag and Horning [2] define *sufficient-completeness* and *consistency* of an algebraic specification in terms of existence and uniqueness of axioms in the specification. Jones [3] defines *bias* for model-based specifications as the property of nonuniqueness of representation within the model. Yue [8] gives a definition for *completeness* of a specification, in terms of the satisfaction of a set of explicitly stated goals. He also defines *pertinence*, a property related to bias. Nicholl [5] defines the concept of *reachability* for model-based specifications as the ability to reach every consistent state by some sequence of operations, and plans to study other features of specifications, including bias.

This current research work grew out of studies within the Software Engineering Laboratory (SEL) of NASA Goddard Space Flight Center, which has been monitoring the development of ground support software for unmanned spacecraft since 1976. Our goal is to improve the quality of software specifications within the SEL. On the realization that existing specification languages were inappropriate for use by programmers at NASA, we developed the executable specification language PUC (pronounced POOK), designed to be used with Ada in this environment [7].

The executability of specification languages like PUC had the disadvantage that much detail had to be included in specifications, limiting the creativity of the implementor and ruling out some possibly good designs. Hence, instead of looking at the language problem, now we are looking at this problem itself, the so-called 'implementation bias' in specifications. The area of bias in specifications is largely unexplored but is important. In fact, the problem of bias is mentioned in several works, including both description and critiques of specification methods.

1.1 Definitions

Some related concepts are defined.

Attribute An *attribute* of a product is a required or desired feature of the product, its environment, or its development process.

We use 'attribute' instead of the more customary 'requirement', because the latter is associated with mandatory features that are described in the initial phases of development.

Specification A *specification* of a product is a description of a set of its attributes.

Under this definition, both the *requirements document* and the *preliminary design document* of the waterfall model are specifications.

Solution set The *solution set* of a problem is the set of all products that solve the problem, regardless of the specification.

1.2 The Problem of Bias

An ideal specification is general and precise enough so that a software system satisfies the specification if and only if it solves the problem at hand. This view is too optimistic, because there can be many solutions to the real world problem that do not even involve software. In practice, we only need that software systems satisfying the specification be solutions, and that no substantial class of solutions does not satisfy the specification.

A specification is biased if it arbitrarily favors some implementations over others. Biased specifications can overly constrain the solution 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.

One of the main problems of not having a good definition of *bias* is that it is sometimes confused with intended constraints in the solution set. For example, a designer may want to favor some realizations over others for compliance with some programming techniques that are customary at that site. In fact, we argue that much of what has been called bias is simply a manifestation of design decisions, that purposely constrain the solution set. Of course, we also have many specifications that are indeed biased.

2 A Model of Bias

We present a framework to discuss bias, based on a classification of the attributes of the product being specified and the process of creation of attributes.

2.1 Classes of Attributes

We classify the attributes of a product with respect to their inclusion in the specification. The main criteria we consider are explicitness and origin.

2.1.1 Explicitness

An attribute is *explicit* if it is present in the specification; otherwise, it is *nonexplicit*. Nonexplicit attributes are further classified in four classes.

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

Implied attributes are logical consequences of other attributes.

Absent attributes are requirements unintentionally omitted in the specification. These are not part of every product in the application domain.

P. Straub
Univ. of Maryland

Fictitious attributes [4] are not attributes at all, but assumptions made by the reader of the specification: the reader believes that they are either implicit, implied or absent attributes.

2.1.2 Origin

An explicit attribute is *new* with respect to a certain specification stage if it is first made explicit at that stage; otherwise, the attribute is *inherited* from previous stages.

In an ideal setting all attributes new in a specification stage are the consequence of design decisions taken at that stage. However, nonexplicit attributes in the previous specification usually induce the specifier of the current stage to introduce extra attributes. Besides, some attributes may be imposed by the limitations of the specification method and language used. This motivates the following classification of new attributes with respect to their origin.

Designed attributes are the consequence of design decisions taken at the current specification stage. They are purposely set to guide the implementation process and constrain the solution set.

Explicatory attributes are created by making explicit attributes that are implicit in, implied by, or absent from previous stages.

Imposed attributes are those imposed by the limitations of the specification method and language used.

For example, a method may accept only "complete" specifications (as defined by the language), which leads to introduce attributes to satisfy the rules of the language.

Extraneous attributes are created by making explicit fictitious attributes.

For example, a fictitious attribute seen by the designer in a requirements document may introduce explicit constraints in the design document.

2.2 The Nature of Bias

The process of refining successive specifications makes explicit attributes that were previously implicit, implied, or absent. This process also makes explicit design decisions taken at the current stage. Unfortunately, it also makes explicit fictitious attributes (i.e., creates extraneous attributes[1]) and creates imposed attributes (Figure 1). This leads to the definition of bias in terms of the origin of the attributes described in a specification.

[1] Extraneous attributes lead to errors and constraints in the solution set; here we are studying only the constraints.

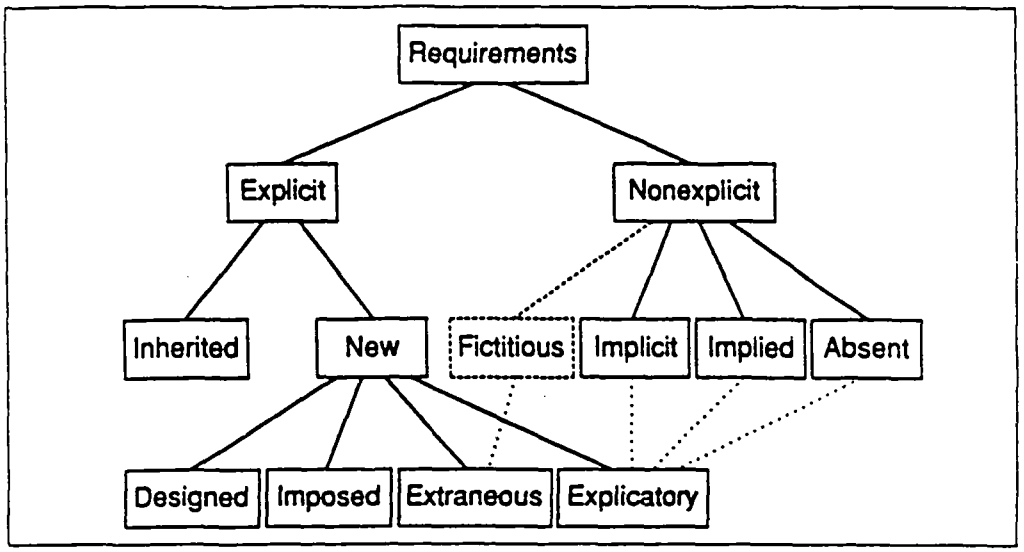


Figure 1: Classification of attributes. Fictitious attributes are shown with segmented line, because they are not real attributes but misconceptions. Dotted lines show the origin of new explicatory and extraneous attributes.

Definition. A specification containing extraneous or imposed attributes 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 attributes or the limitations imposed by the specification method. The consequences are that the set of possible solutions 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.

Bias content in a specification cannot be measured directly, because bias is defined in terms of the origin of attributes which is usually uncertain. Furthermore, bias is relative to the application domain and the software engineering environment, because the domain and environment define what is implicit.

The relative nature of bias is an essential characteristic. It stems from the existence of nonexplicit attributes and the inherent uncertainty with respect to those attributes. As long as there are nonexplicit attributes, there will be doubt about these attributes and hence possibility of bias. Furthermore, making explicit all implicit attributes of a certain domain and environment still leaves two sources of bias: restrictions on the method and languages, and absent attributes.

2.3 Example

Assume an environment in which all programs are written in a particular programming language. In this environment 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 SEL where software specifications for satellite dynamic 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 attributes. Before Ada was introduced, the use of FORTRAN was implicit. After that, the language used had to be decided: assuming a FORTRAN implementation was a fictitious attribute.

3 Current Research

We are improving the model presented in this paper in several aspects.

Formalization One weakness of our model as presented here is that we do not formalize the concept of 'attribute'. Moreover, we define 'specification' as a set of attributes, disregarding dependencies among attributes. At least two kinds of dependencies are relevant: attributes defined in terms of other attributes, and origin relationships among attributes.

To address this problem, we have develped a formalism to write specifications that is flexible and extensible enough to include information about the specification itself (e.g., origin information). Within the system, called Extensible Description Formalism (EDF), attributes are defined as mappings from objects to values; objects are represented by extensible polymorphic records whose fields are the attribute names. EDF can represent both functional dependencies of attributes and also attributes defined as aggregations of several attributes. Origin information is stored by representing all attribute values as objects that have an *origin* attribute and a *content* attribute.

We developed a prototype of EDF and used it in the context of classification of reusable software components. We are currently developing a complete version based on a formal specification of the language [6].

Measuring Bias In this work we have not provided a characterization of biased specifications. Because of the relative nature of bias we cannot develop a precise

metric of bias, but we can define approximate metrics, based on origin information explicitly recorded in a specification.

An important feature of EDF is that it is possible to compare two specifications defining some *distance* from one specification to another. There is a predefined comparator function to estimate the adaptation effort in the context of reuse of software components, and it is possible to define other comparator functions.

We can measure bias comparing the distance between two succesive specification stages. If we use the predefined comparator function we get a gross upper bound on bias (as if all attributes new to the second stage were bias). On the other hand, by defining a comparator function that uses origin information, we expect to provide a reasonable estimate of bias introduced in the second stage.

Bias Propagation Our model does not explain how bias propagates, because we have defined bias in terms of new attributes. Strictly speaking, within our model no inherited attribute is bias. Since we want to measure bias content in a specification, we have to consider those attributes whose origin include extraneous or imposed attributes. For example, if a design decision is taken consistently with some inherited attribute that was extraneous when created, then this decision has some form of bias too.

Bias and Software Defects Our model describes the origin for software attributes, and defines bias as the existence of some attributes with 'illegitimate' origin. The reader can realize that these illegitimate origins are also the cause of software defects.

Software defects are classified in three groups: *errors* are conceptual misunderstandings, *faults* are concrete (explicit) manifestations of errors in documents, and *failures* are manifestations of faults during execution.

There is an intimate relashionship between errors and fictitious attributes. and between software faults and bias. In a sense, bias is like a very minor fault that instead of leading to failures, leads to inefficiencies. The consequence of this is that methods to avoid bias (e.g., making explicit implicit requirements) will also avoid software defects.

4 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 concepts of design constraint and design decision, so that the presence of designed attributes 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 attributes (e.g., materials, manufacturing methods).

In this paper we presented a model to describe the nature of bias and distinguish bias from designed attributes and other attributes in a specification. This model is based on a classification of all the attributes 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 attributes it does not lead to any definite method to detect bias. However, the model does explain both the relative and unavoidable nature of bias. Moreover, because the model explains the origin of bias, it provides insight into bias avoidance.

Our goal is to improve the quality of the specifications by removing bias and including all relevant implementation-oriented information. To achieve this goal we need to tell bias from designed attributes. This requires information on the origin of the attributes, which is usually unknown. Hence, we have developed a formalism in which origin information can be recorded, as a generalization of the common practice of tracing design documents and actual code back to the original statement of requirements.

Acknowledgements

Thanks to Sergio Cárdenas-García and Eduardo Ostertag for their helpful comments. The Extensible Description Formalism (EDF) was defined jointly with Eduardo Ostertag, who is also the implementer.

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] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

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

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

[5] Robin A. Nicholl. Unreachable states in object-oriented specifications. *IEEE Transactions on Software Engineering*, 16(4):472–477, April 1990.

[6] Pablo A. Straub and Eduardo J. Ostertag. Semantics of the Extensible Description Formalism. Technical Report CS-TR-2561, UMIACS-TR-90-137, University of Maryland, Department of Computer Science, November 1990.

[7] Pablo A. Straub and Marvin V. Zelkowitz. PUC: A functional specification language for Ada. In *X International Conference of the Chilean Computer Science Society*, pages 111–122, Santiago, Chile, July 1990.

[8] Kaizhi Yue. What does it mean to say a specification is complete? In *Fourth Int'l Workshop on Software Specification and Design*, pages 42–49, Los Alamitos, California, 1987. CS Press.

VIEWGRAPH MATERIALS

FOR THE

P. STRAUB PRESENTATION

6269-0

Bias and Design in Software Specifications

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

Contents

Introduction
Importance of specifications

Life-cycle models consist of refinement of succesive specification stages.

Specification: description of a set of requirements.

'Requirement' is used in all stages, not just the first.

Staged specifications imply

- errors in previous stages are costly
- product quality depends on specification

We need high quality specifications.

Introduction

We want specifications that are...

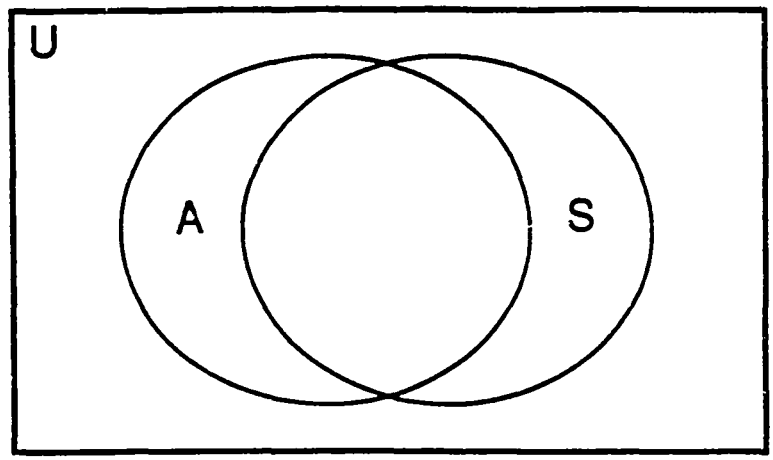
- abstract
- complete
- consistent
- correct
- reusable
- traceable

} our focus

- concise
- executable
- feasible
- formal
- modifiable
- realizable
- structured
- verifiable

P. Straub
Univ. of Maryland

Introduction
Solutions vs. specified products



U = product universe

A = acceptable products (solutions)

S = specified products

$S - A$ = specified unacceptable products

$A - S$ = solutions not specified

Ideally: $S = A$

Needed: $S - A = \emptyset$

Desired: $A - S$ is *small*

P. Straub
Univ. of Maryland

Introduction
The *what* and *how* dilemma

Typical rule to avoid overspecification

> Specify *what* the system should do,
> not *how* to do it.

But *what*'s and *how*'s depend on viewpoint.

What: something already fixed

How: an option

How's become *what*'s.

Confusion creates underspecification.

P. Straub
Univ. of Maryland

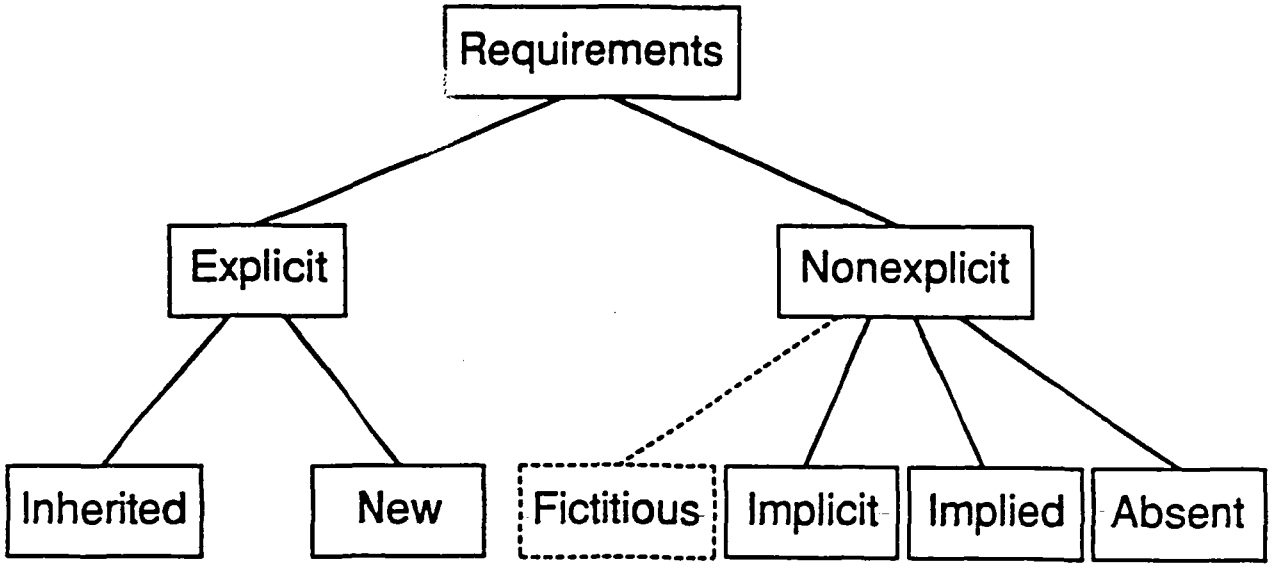
Classification of requirements
Explicitness

Requirements of a product are classified as

- Explicit: written in the specification
  - Inherited: comes from previous stages
  - New: created at this stage

- Nonexplicit: not written
  - Fictitious: not a requirement, but a misconception
  - Implicit: belongs to all products
  - Implied: consequence of other requirements
  - Absent: unintentionally omitted

Classification of requirements
Explicitness



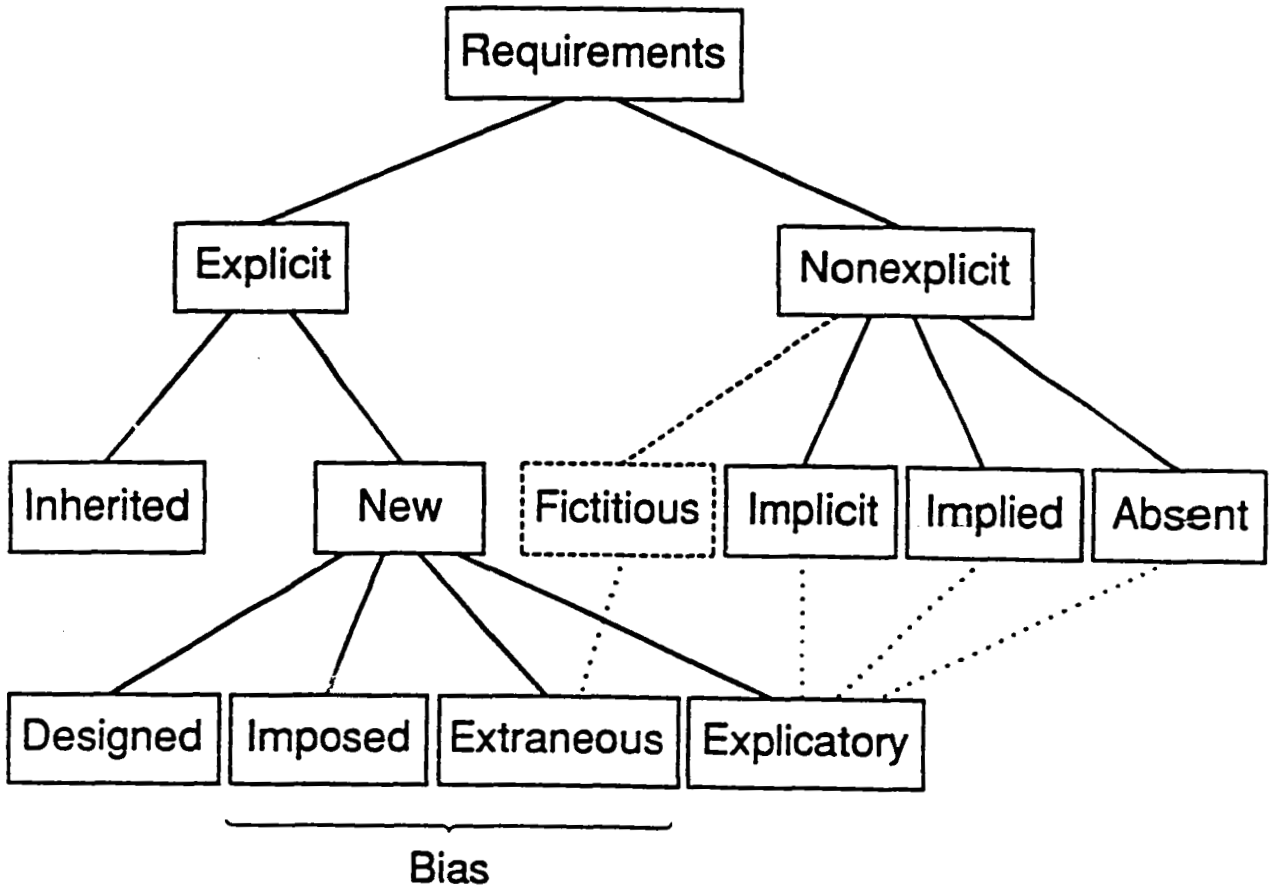
Classification of requirements
Origin of new requirements

New requirements are classified as

- Designed: restriction on purpose
- Imposed: restriction of method or language
- Extraneous: makes explicit a fictitious requirement
- Explicatory: makes explicit a nonexplicit requirement

P. Straub
Univ. of Maryland

Classification of requirements
Creation of new requirements



The nature of bias
Definition

Definition

> A specification containing extraneous or imposed requirements is *biased*.

Origin of bias

- wrongful interpretation of nonexplicit requirements
- limitation imposed by method or language

Consequences of bias

- solutions not specified
- adoption of a nonoptimal solution

P. Strauss
Univ. of Maryland

The nature of bias
Essential limitations

Bias is not an absolute property.

Bias depends on origin.

Bias depends on application domain and environment, because of different implicit requirements.

Bias cannot be completely eliminated.

P. Straub
Univ. of Maryland

The nature of bias
Example (at NASA/GSFC)

Before introduction of Ada, FORTRAN was implicit.

Specifications had many FORTRAN-oriented requirements.

During first Ada project, the specifications had to be rewritten.

After introducing Ada, assuming FORTRAN was a fictitious requirement.

Conclusions
Other Considerations

- Formal definition of 'requirement'.
- Method to find bias.
- Formalism to write specifications with attributes (e.g., origin of requirements).

Conclusions
Contributions

A theory of bias

- classification of requirements
- origin of requirements
- precise definition of bias
- bias is inherent to specifications

The Extensible Description Formalism (EDF)
a language to

- describe requirements and their attributes
- compare specifications
- measure bias

Bias in relationship with software defects:

- errors $\leftrightarrow$ fictitious requirements
- faults $\leftrightarrow$ bias

Conclusions
Next Steps

Try these ideas measuring bias in a specific project.

Extend this theory to explain creation of software defects.