Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A STUDY OF THE APPLICATION OF FORMAL SPECIFICATION METHODS FOR FAULT-TOLERANT SOFTWARE

Ann Tsu-Ann Tai

December 1988
CSD-880100

UNIVERSITY OF CALIFORNIA

Los Angeles

A Study of the Application of Formal Specification Methods for Fault-Tolerant Software

A thesis submitted in partial satisfaction of the

requirements for the degree of Master of Science

in Computer Science

by

Ann Tsu-Ann Tai

1986

The thesis of Ann Tsu-Ann Tai is approved.

John P.J. Kelly

David A. Rennels

Daniel M. Berry

Algirdas Avižienis, Committee Chair

University of California, Los Angeles

1986

TABLE OF CONTENTS

LIST OF FIGURES

page

ACKNOWLEDGMENTS

I would like to thank my committee members, Professors Algirdas Avižienis, Daniel Berry, David Rennels, and John Kelly for serving on my thesis committee with such enthusiasm.

I wish to express my sincere thanks to Prof. Avižienis for his guidance, advice and encouragement that have contributed significantly to the preparation of this thesis.

Special thanks to Prof. Kelly for his guidance and support of my research that has made this thesis possible.

Prof. J.V. Guttag of MIT, Dr. J.J. Horning of DEC Systems Research Center, and Prof. J.M. Wing of Carnegie-Mellon University have spent much time and sent much electronic mail to help me to understand Larch. Very special thanks to them.

Thanks to Prof. Berry for reading the manuscript drafts so carefully and helping me to state my meaning better.

Thanks to members of the UCLA Dependable Computing and Fault-Tolerant Computing research group, especially Jean-Paul Blanquart, Yasuhiko Hatakeyama, Rong-Tsung Lyu, Barbara Swain, Kam-Sing Tso, and Bradford Ulery for invaluable discussions that improved the quality of this work.

I must give my heartfelt thanks to my husband Kam-Sing for his love and dedication.

This work was supported by the Federal Aviation Administration Advanced Computer Science Program (via NSF Grant MCS81-21696), under the direction of A. Avižienis, Principal Investigator, and the National Aeronautics and Space Administration, Contract NAG1-512, under the direction of A. Avižienis, Principal Investigator, and J.P.J. Kelly, Co-Principal Investigator.

ABSTRACT OF THE THESIS

A Study of the Application of

Formal Specification Methods for Fault-Tolerant Software

by

Ann Tsu-Ann Tai

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor Algirdas Avižienis, Chair

The main purposes of this research are to discuss the importance of formal specifications and to identify a number of promising specification techniques of Larch. Common deficiencies of informal specification aspects are identified with reference to an English specification used in a fault-tolerant software experiment. Criteria are established for evaluating the practical potential of formal specification techniques for fault-tolerant software. A number of specification techniques for describing an application in the fault-tolerant software are surveyed and evaluated with respect to these criteria. Basic features of Larch which make it feasible to use it to specify fault-tolerant software are presented. The difficulties encounted and experiences gained during the learning activities and the specifying process are discussed. Common errors that occurs during the specification process are identified and classified with the help of case studies. The distinctive style of Larch specifications is studied. How to improve the comprehensibility of Larch specifications is discussed. Finally the limitations of the present Larch specification language are indicated.

CHAPTER 1

INTRODUCTION

An ideal software specification provides a description of the desired external behavior of the software without describing or constraining its internal implementation. Specification is crucial in software development as it has been estimated that the cost to fix an error detected during operation can be 1000 times greater than the cost to fix the same error had it been detected during the specification phase [Boeh81]. Specifications have to be complete, consistent, unambiguous and correct. Past experiences in software development show that faults due to an inadequate specification are very difficult and expensive to correct when they are discovered in later phases of software development.

A growing concern on the reliability of computer systems is design faults. Design diversity [Aviz82] is an approach to tolerate design faults. In this approach hardware and software elements are not copies, but are independently designed from an initial specification to meet a system's requirements. Different designers and design tools are employed in each effort. It is hypothesized that although errors may occur in some of the elements, a majority of good results can be found because faults are independent, if they exist.

Multi-Version Software (or N-Version Programming) [Aviz77, Aviz85b], the generation of $N \geq 2$ software "versions" from the same specification, is an approach to tolerate software faults by design diversity. The specification is to state the functional

requirements completely and unambiguously, while leaving the widest possible choice of implementations. Thereafter, N independent versions are independently written by N programming teams that do not interact. Since the versions are written independently, it is hypothesized that the probability that they contain similar errors is greatly reduced, i.e., errors in their results are largely uncorrelated. Since the implementation of the software versions are based on the same specification, the specification is effectively the hard core of the approach. A possible way to get rid of the hard core is to have one more level of design diversity, that is, using multiple specifications to specify the same functional requirements. A specification oriented Multi-Version Software experiment conducted at UCLA [Kell82, Kell83] has demonstrated that specification faults are the most serious and there is a need for improvement in specification techniques.

In the next chapter the deficiencies of the English specification of the Redundant Strapped Down Inertial Measurement Unit (RSDIMU) used in the Second Generation Experiment [Kell86] are discussed and the importance of using formal specifications is highlighted. Various formal specification languages have been considered for formally specifying RSDIMU. Chapter 3 presents the rationale for choosing LARCH. Chapter 4 contains an overview of LARCH, and Chapter 5 records the experiences gained in the learning process of LARCH. Chapter 6 discusses the specification of RSDIMU in LARCH. The specification itself can be found in the Appendix.

CHAPTER 2

THE DEFICIENCIES OF INFORMAL SPECIFICATIONS

2.1 The English Specification of RSDIMU and Its Deficiencies

A large scale experiment was conducted in the summer of 1985 to determine the effect of fault-tolerant software techniques under carefully controlled conditions [Kell86]. An application was programmed by 20 teams of two students each from four universities* working in a carefully controlled programming environment. The application, called the Redundant Strapped Down Inertial Measurement Unit (RSDIMU), had been chosen as a realistic example from the avionics area. It was specified in English by the staff of Charles River Analytics, Cambridge, MA [Redu85]. The specification consists of 64 pages of English description and was augmented by 250 question and answer pairs and 10 Announcements (about 60 pages long). The questions were raised by the programmers during the development of the program versions. Answers to those questions were broadcast every programmers. The Announcements are used to correct errors in the initial specification. In the course of the software development and the preliminary evaluation of the resulting programs we found that the specification was grossly inadequate.

Table 1 [Meye85] lists seven classes of deficiencies in natural language specification that we have found to be both common and particularly damaging to the quality of requirements. The followings show some of these deficiencies in the

*They are the University of California, Los Angeles, the University of Illinois at Urbana-Champaign, North Carolina State University, the University of Virginia.

RSDIMU Specification:

Noise:	The presence in the text of an element that does not carry information relevant to any feature of the problem.
Silence:	The existence of a feature of the problem that is not covered by any element of the text.
Overspecification:	The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.
Contradiction:	The presence in the text of two or more elements that define a feature of the system in an incompatible way.
Ambiguity:	The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.
Forward reference:	The presence in the text of an element that uses features of the problem not defined until later in the text.
Wishful thinking:	The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

Table 1: The Seven Sins of the Specifier

1. An instance of *noise* appears on page 32 of the specification. Paragraph 2 says "If a specific accelerometer on a given face is declared to be failed prior to the invocation of the program, then that face is not used in the edge vector test." In the beginning of the next paragraph, it states "If a single sensor on a face fails, the face cannot be used in the edge vector test." When such a statement is first encountered, the reader may think it brings new information, but upon

closer examination, he realizes that it only repeats known information in new terms. The reader must thus ask himself nonessential questions, which divert attention from the truly difficult aspects of the problem. *Noise* is also a source of contradiction and ambiguity.

2. The specification is *silent* in some critical points. In Appendix C, Least Square Vehicle Acceleration Estimation, the specifier tries to define the computations that are to be performed by using the various sensor subsets. It says "If the x-accelerometer on face B and y-accelerometer on face C are determined to be failed, then the best estimate of the vehicle acceleration is computed by performing the computations above with 6*3 partitions of the matrix C obtained by deleting the third and sixth rows." Thus the y-element on face B and x-element on face C are still among the six rows. But which frame are they in? It is not specified. Most programmers did not have enough engineering background and made the assumption that specific force in a good accelerometer isolated from a bad face should remain in the Sensor-Frame as those in a good face do. Not until the Unit Testing Stage did the specifier give a clear definition in Announcement VII. 30 questions concerning this inadequacy are the strong evidence of the sin of the *silence*.

   Another example of *silence* is shown in appendix D regarding the Edge Vector Test. The specification only asserts that 3 common threshold test failures is the sufficient condition for determining a bad face but keeps silent on what the necessary condition is. About 20 questions regarding this *silence* were raised. (The final solution from RTI was still unreasonable.)

3. The specification *overspecifies* the Least Square Algorithm. Instead of specifying the requirement it gives the solution which uses matrix inversion.

In fact it is not a very good solution unless the residual correction method is used. Overspecification is especially bad in multiple version software because it reduces the degree of design diversity as extra details lead independent programmers to follow a similar structure and increase the probability of similar errors.

4. The definition of LINOUT says, "This will hold real values representing the linear acceleration component of sensor....". However, page 60 defines that $\hat{f}$ which is an approximation of LINOUT in Instrument Frame is the specific force. Together with the equations

$$\text{LinearAcceleration} = \text{Offset} + \text{Slope} \times \text{Voltage} \quad \text{(Sec. 2.3.3)}$$

$$\hat{a} = T_{NI}\hat{f} + g^N \quad \text{(Appendix C)}$$

a *contradiction* occurred and similar error followed.

5. *Ambiguities* are very common in the specification. One typical example is in the definition of LINOUT. It states that some exception handling ought to be done on these "failed sensors". It is possible to interpret the words "failed sensor" as either the sensor failed prior to the flight or the sensor failed during the flight. The consequence of the different interpretations was that multiple versions disagreed on LINOUT and badly degraded the dependability of the fault-tolerant software system.

6. The specification uses quite a few *forward references*. Not all of them are bad but some implicit ones really confuse implementors. For instance, the Specification defined an input variable NORMFACE at page 12 without defining its usage and mentions a variable f_{AS} after 46 pages without explicit indication that they refer the same thing. That is why so many questions were

raised about NORMFACE.

7. The Specification defines an operational status ANALYTIC on page 21. The definition is "Exactly 3 instrument values are available from which to compute the 3 components of acceleration in the Navigation Frame of Reference." However, on page 19, the definition of SYSSTATUS says "(SYSSTATUS) is TRUE if at least two faces are completely operational and their edge vector satisfies the threshold test. It is FALSE otherwise. If the value of this variable is FALSE, set .... all acceleration estimate status indicators to UNDEFINED ....." If exactly 3 sensors are working in the system, there is at most one face completely operational and its edge vector satisfies the threshold test. In that case, the acceleration estimate status should set to UNDEFINED according to definition of SYSSTATUS. Therefore, the ANALYTIC status will never appear in the acceleration estimate status of the output variable BESTEST. That is an example of *wishful thinking*.

2.2 The Necessity of Formal Specifications

The obvious advantage of design diversity is that reliable computing does not require the complete absence of design faults but only that those faults not produce similar errors in a majority of the designs. Analysis of the twenty versions has provided some insight into the causes of similar errors. The most prevalent cause has shown to be the specification. Boundary conditions, exceptions, and other such design and coding errors appear to be both less frequent and less correlated. This observation strongly supports multi-version software as a means of diminishing the number of faults introduced during the design and coding phases of development. This observation also shows that Multi-Version Software demands a high quality

specification for similar error avoidance.

In a fault-tolerant software system, the most critical condition for the independence of design faults is the existence of a complete and accurate specification of the requirements that are to be met by the diverse designs. This is the hard core of this fault tolerance approach. Latent defects, such as inconsistencies, ambiguities, and omissions in the specification are likely to bias otherwise entirely independent programming or logic design efforts toward related design faults [Aviz85b].

The RSDIMU English Specification has many deficiencies. Although well written requirements are obviously preferable to poorly written ones, we doubt that they alone solve the problem. In our view, natural language descriptions of any significant system, even good quality, exhibit deficiencies that make them unacceptable for rigorous software development. The most promising approach to the production of the initial specification is the use of a formal, very-high-level specification language [Lisk79, Parn79, Mell79].

A specification is formal if it is written entirely in a language with an explicitly and precisely defined syntax and semantics. Its purpose is to provide a mathematical description of the concept; and the correctness of a program is established by proving that it is equivalent to the specification. There are advantages in using formal, rather than informal specifications. Formal specifications can be studied mathematically while informal specifications cannot. Formal specifications can also be meaningfully processed by a computer. Certain forms of inconsistency or incompleteness in the specification can be detected automatically. Since this processing can be done in advance of implementation, it should prove to be a valuable aid to program design.

Because it is difficult to construct specifications using informal techniques, such as English, specifications are often omitted, or are given in a sketchy and incomplete manner. This may explain some deficiencies illustrated in the previous section. Formal specifications techniques can provide a concise and well-understood specification or design language, which should reduce the difficulty of constructing specifications.

Problems arise if the specification is ambiguous: that is, if it fails for some reason to capture a unique concept so that two programs with different computational properties both satisfy the specification. The existence of ambiguity often is not realized, and instead the ambiguity is resolved in different ways by different people. This was a common scenario in our second generation multi-version software experiment. Formal specifications are less likely to be ambiguous than informal ones because they are written in a well-defined language. Also, the meaning of a formal specification is understood in a formal way, and therefore ambiguities are more likely to be recognized.

Thus, formal specification techniques are valuable aids to fault-avoidance and fault-removal in the development of reliable software.

CHAPTER 3

FORMAL SPECIFICATION AND FAULT-TOLERANT SOFTWARE

Significant progress has occurred in the development of formal specification languages in the past few years. Examples are the CLEAR specification language developed at Edinburgh University and SRI International [Burs81], The "M" specification language developed at Oxford University [Meye84], The Ina Jo specification language developed at SDC [Loca80], The OBJ2 specification language developed at SRI International [Futa85], and the Larch family of specification languages developed at MIT, Xerox PARC and DEC [Gutt85a].

3.1 Criteria for Selecting Specification Languages

The selection of a specification language for Fault-Tolerant Software must satisfy a number of requirements. The criteria described below are practical as well as theoretical considerations [Lisk77, Kell82].

1. *Formality.* The syntax and semantics of the language in which the specifications are written must be fully defined. A specification method should be formal, that is, specifications should be written in a notation which is mathematically sound and provable. The formality plays a fundamental role in fault-avoidance in the construction of reliable software.

2. *Constructibility.* It must be possible to construct specifications without undue difficulty. Two facets of the construction process are of interest here: the

difficulty of constructing a specification in the first place and the difficulty in knowing that the specification captures the concept.

3. *Comprehensibility.* A person trained in the notation being used should be able to read a specification and then, with a minimum of difficulty, reconstruct the concept which the specification is intended to describe. Properties of specifications which determine comprehensibility are size and lucidity. Comprehensibility is a necessary condition for avoidance of similar errors.

4. *Minimality.* It should be possible using the specification method to construct specifications which describe the interesting properties of the concept and nothing more. The properties which are of interest must be described precisely and unambiguously but in a way which adds as little extraneous information as possible. In particular, a specification must say what functions a program should perform, but little, if anything, about how the function is performed. One reason this criterion is desirable is because it maximizes the design diversity of fault-tolerant software.

5. *Wide Range of Applicability.* Associated with each specification technique there is a class of concepts (domain of applicability) which the technique can describe in a natural and straightforward fashion, leading to specification satisfying the criteria for constructibility and comprehensibility. Concepts outside of the domain can only be defined with difficulty, if they can be defined at all. Clearly, the larger the class of concepts which may be easily described by a technique, the more useful the technique.

6. *Extensibility.* It is desirable that a minimal change in a concept should result in a similarly small change in its specification. This criterion especially

impacts the constructibility of specifications.

The second and third criteria address the fundamental problem with specification techniques -- the difficulty in using formal specification techniques.

3.2 Rationale For Using Larch

The techniques of specification fall into five categories [Lisk77]:

1. use of a fixed domain of formal objects, such as sets or graphs;

2. use of an appropriate, but otherwise arbitrary, formal domain;

3. use of a state machine model;

4. use of an implicit definition in terms of axioms;

5. use of an implicit definitions in terms of algebraic relation.

Larch is an algebraic method. The algebraic specification method provides a method for specifying a type by defining an axiom set for the type. The method is largely free from any representational or operational content, thus avoiding undue bias on the subsequent implementation. Thus, the algebraic specification is feasible to fault-tolerant software.

Each Larch specification has two parts, written in different languages. One part is written in a language derived from a programming language and another is written in a language independent of any programming language. The former is called Larch interface language, and the latter is called Larch Shared Language. Larch interface languages are used to specify program units (e.g., procedures, modules, types). Their semantics is given by translation to predicate calculus.

Abstractions appearing in interface specifications are themselves specified algebraically, using the Larch Shared Language.

Some important aspects of the Larch family of specification languages are [Gutt85b]:

1. *Composability.* The Larch Shared Language is oriented towards the incremental construction of specifications from other specifications. This bottom-up construction method breaks down a large piece of abstraction into smaller ones, thus easing the difficulty of writing a large specification. The feature of composability also provides extensibility of a specification.

2. *Emphasis on Presentation.* To make it easier to read and understand specifications, the composition mechanisms in the Larch Shared Language are defined as operations on specifications, rather than on theories or models. This supports the criterion of comprehensibility.

3. *Semantic Checking.* The semantic checks for the Larch language were designed assuming the availability of a powerful theorem prover. Hence they are more comprehensive than the syntactic checks commonly defined for specification languages. This feature strengthens the formality of Larch.

4. *Availability of Handbooks.* It is inefficient to start each specification from scratch. We need a repository of reusable specification components that have evolved to handle the common cases well, and that can serve as models when we are faced with uncommon cases. The Larch Shared Language Handbook has a rich collection of essential reusable units in Larch, and provides great convenience for constructing specifications. The handbook is the concentrated

essence of abstractions that experienced specifiers found useful.

5. *Incompleteness.* Most specifications of Larch can be partial. Sometimes incompleteness reflects abstraction from details that are irrelevant for a particular purpose; sometimes it reflects an intentional choice to delay certain design decisions. The incompleteness which leaves wider range of implementation choices to programmers is advantageous to design diversity in fault-tolerant software system.

Thus, Larch is a qualified candidate to assess the promise of formal specification methods for dealing with fault-tolerant software.

Other good reasons for choosing Larch are:

1. *Support of the authors of Larch.* Drs. Guttag, Horning, and Wing were willing to provide advice and support (which is documented later).

2. *Availability of the Good Documentations. Larch in Five Easy Pieces* and other Larch literature were available for us, which contain introductory, motivating, and reference information.

CHAPTER 4

AN OVERVIEW OF LARCH

4.1 Introduction

The use of formal specification in software development offers significant advantages. Although there are many theoretical research activities, practical experience is limited. The Larch Project, a research project intended to aid in putting formal specification into practical use, is developing both a family of specification languages and a set of tools to support their use [Gutt80, Gutt83, Gutt85a, Gutt85b].

Each Larch specification has one component written in a language derived from a programming language and another component written in a language independent of any programming language. The former is called the *Larch Interface Language* and the latter is called the *Larch Shared Language.*

The predicate-oriented interface languages are used to describe the intended behavior of procedures. Abstractions are formulated in the Shared Language. Descriptions given in the interface languages are given in terms of those abstractions and might also include descriptions of error reactions and implementation limits.

4.2 The Larch Shared Language

Similar in appearance to many algebraic specification languages, the Shared Language can be used for specifying abstract data types, but its focus is on specifying

smaller entities or properties (such as commutativity, group theory, and generic properties of container-like types). Such entities are expressed as independent, tractable, and reusable building blocks.

The Shared Language offers a simple, syntactic approach to modularization and composition. Units of specifications, called traits, are combined by syntactic inclusion; inclusions can be equipped with renaming rules. Traits are never explicitly parameterized; the renaming mechanism makes any entity of a trait a potential parameter. The meaning of a trait is first-order theory. It is obtained as the conservative union of the theories associated with included traits and the set of local axioms of a trait. The local axioms are expressed as first-order, quantified equations.

The following is a piece of a *Larch/Pascal Interface Language* specification. The terms in *italic* are the names of traits defined in the Shared Language part.

function bagChoose(b:bag; var e: integer): boolean
    modifies at most [e]
    ensures if ¬*isEmpty(b)*
        then bagChoose & *count(b,* e_{post}*)*>0
        else ¬bagChoose & **modifies nothing**

The basic unit of Shared Language is a *trait* which introduces terms and specifies their properties.

The *functional structure* of a trait is as following:

Name: **trait**
    [includes|assumes|imports] (names of traits prior defined)
    [introduces] (names of new operators)
    [constrains|asserts] (quantified equations)
    [**converts**] (names of operator which have sufficient axioms)

The **imports** statment says that the importing trait is a conservative extension of the theory of the imported trait. The **includes** statment indicates that the including

trait intends to inherit the imported traits' operators and to further constrain them. We use importation when we can incorporate a theory unchanged, and inclusion when we cannot. The **assumes** statment is similar to **includes** statment but assumptions can be discharged by explicit inclusion.

The **introduces** statment declares a set of new operators (function identifiers), each with its *signature* (the sorts of its domain and range).

The **constrains** statment constrains operators by means of equations that relate terms containing them. In general, each equation involves several operators, and an operator may appear in several equations. The **asserts** statment is used when **constrains** would supply no information (e.g., there are too few operators in the trait to build any quantified equations).

The **converts** statment indicates that this trait is intended to contain enough axioms to define the operators introduced.

Here is an example which shows us the *reusability* of trait:

Container: **trait**
    introduces
        new: → C
        insert: C,E → C
    asserts C **generated by** [new, insert]

Size: **trait**
    assumes Container
    imports Cardinal
    introduces size: C → Card
    constrains size **so that**
        size(new) = 0

AdditiveSize: **trait**
    assumes Container
    includes Size

constrains size, insert **so that for all** [c:C,e:E]
size(insert(c,e)) = size(c) + 1
converts [size]

It is noted that AdditiveSize has the **converts** part while Size does not. This is because Size leaves intentional incompleteness for a property of *size* whereas AdditiveSize completes its definition.

The incompleted operator *size* can be reused in different including traits which give it complete but different properties from these obtained in AdditiveSize. E.g. *size* can be given another definition in another including trait:

includes Size **with** [size1 **for** size]

....

size1(insert(c,e)) = size1(c) + **if** e ∈ c **then** 0 **else** 1

which counts the number of kinds of different elements in a set. By *reusing* existing traits, specifiers will save time and avoid errors.

The following is a group of traits which shows the *composability* of Shared Language:

Relation: **trait**
introduces # ® #: T,T → Bool

Reflexive: **trait**
includes Relation
asserts for all [x: T]
x ® x

Transitive: **trait**
includes Relation
asserts for all [x,y,z: T]
((x ® y) & (y ® z)) => (x ® z)

ReflexiveTransitive: **trait**
includes Reflexive, Transitive

PartialOrder: **trait**
imports ReflexiveTransitive **with** [≤ **for** ®]

They show how complicated traits are composed of simple ones and what the incremental structure looks like. The Larch languages are designed for incremental construction of specifications from other specifications.

New traits are unlikely to have as much structure as is present in the various specializations of Container. This kind of structure tends to come after a large number of related traits have been written and regularities recognized, or when the abstraction represents a well-studied mathematical system.

4.3 Larch Interface Languages

Larch interface languages are those specifications that actually describe program units to be implemented. The role of the Larch Shared language traits is to define the theories that give meaning to operators that appear in the interface specifications.

Each Larch interface language is designed for a particular programming language. Everything from the modularization mechanisms to the choice of reserved words is influenced by the programming language. Larch/Pascal and Larch/CLU are the only two moderately well-developed Larch interface languages to date.

Both Larch/Pascal and Larch/CLU support the specification of data and procedural abstractions. For each language, we consider one data abstraction, containing several procedural abstractions.

In both Larch interface languages, a specification of a data abstraction has three parts. The first is a header giving the type name and the names of the externally visible routines. The second is an associated trait and a mapping from the types in the data abstraction to sorts in the trait. The third comprises the interface specifications for each routine (procedure or function) of the type. A specification of a routine has three parts:

1. A header giving the name of the routine and the names and types of its formals (parameters and returned values).

2. An associated trait providing the theory of the operators that appear in the body.

3. A body stating any requirements the routine's parameters and specifying the effects the routine must have when those requirements are met.

The body of each routine's specification in Larch/Pascal places constraints on proper arguments for calls on the routine and defines the relevant aspects of the routine's behavior when it is properly called. It can be translated in a straightforward way to a predicate over two states in the style of Hehner [Hehn84] by combining its three predicates into a single predicate of the form

```
requires =>
      (modifies predicate &
      ensures predicate).
```

An omitted **requires** is interpreted as **true**.

In the body of a Larch/Pascal specification, as in Pascal, the name of a function stands for the value returned by that function. Formal parameters may appear unqualified or qualified by *post*. An unqualified formal stands for the value of

that formal when the routine is called. A formal qualified by post, for example b_{post}, stands for the value of that formal when the routine returns.

The clients must establish the **requires** clause at each point of call. Having done that, they may presume the truth of the **ensures** clause on return, and that only variables in the **modifies at most** clause are changed. They need not be concerned with how this happens.

The implementors are entitled to presume truth of the **requires** clause on entry. Given that, they must establish the **ensures** clause on return, while respecting the **modifies at most** clause.

The specification of each routine in an interface can be understood without reference to the specifications of other routines--unlike traits, in which the specification constrains the operators by giving the relations among them.

4.4 Two-tiered Specifications

The Larch Shared Language is used to specify a theory rather than a model, and the Larch interface languages are built around predicate calculus rather than around an operational notation. One consequence of these differences is that Larch specifications are less prone to implementation bias.

CHAPTER 5

EXPERIENCES IN LEARNING LARCH

5.1 Difficulties Encounted in Learning Larch

At the beginning of learning the Larch specification techniques, a number of difficulties were encountered.

5.1.1 Lack of Past Experience

Recently there has been a great deal of theoretical interest in formal specifications. However, there has not been a corresponding increase in their use for software development.

The Larch Project is developing tools and techniques intended to aid in the productive use of formal specifications. Nevertheless, even the specification language developers do not yet have much experience with their use in practical software development, and the supporting tools are not yet available [Horn85]. To be the software developer assessing the promise of Larch for dealing with practical problems is not easy.

5.1.2 Insufficiency of Literature

The Larch Project is an effort to test the ideas about making formal specifications useful. The Larch Project is indeed of a research nature. Therefore, the

Larch literature emphasizes the analysis of the attributes of Larch rather than the essence of the usage of the Larch syntax and semantics. The most recent Larch literature is "Larch in Five Easy Pieces" report [Gutt85b]. *Larch in Five Easy Pieces* is written for formal specification researchers and developers who have a certain depth of knowledge about formal specification. *Larch in Five Easy Pieces*, however, is not an easy tool for novices trying to master the specification techniques.

Furthermore, no formal semantics of Larch/Pascal have been published. The developers have been fairly negligent about publishing formal semantics of interface languages [Wing86], and they have not yet written any large specifications in any Larch interface languages [Horn85].

5.1.3 Transition to Practical Applications

Larch in Five Easy Pieces provides fairly numerous examples of Larch traits and exhibits the style of Larch specifications. Nevertheless, almost all those example traits are abstract data types and mathematical primitives. We anticipated that there would be a significant distance between those sample traits and most traits used in fault-tolerant software such as DEDIX or RSDIMU.

Difficulty appeared because the real number data type is used throughout the RSDIMU. An uncountable domain, such as the real numbers, cannot be defined using the algebraic approach [Lisk77].

There are many mapping functions (sensor to face, channel to face, edge to sensor, etc.). How can we define these mapping functions without an intuitive picture or a table? There is neither an analogical example nor a direct clue for these functions in the literature.

5.2.2.1 The First Version of the Real Number Decision Algorithm

The following is the first version of the Real Number Decision Algorithm:

```
% Selects the maximum element of a set

Max: trait
        imports Member, Ordered
        introduces max: C → E
        constrains max so that for all [e:E, b:C]
            ( e ∈ b ) => ( e ≤ max(b) )
        converts [max]


% Selects the minimum element of a set

Min: trait
        imports Member, Ordered
        introduces min: C → E
        constrains min so that for all [e:E, b:C]
            ( e ∈ b ) => ( e ≥ min(b) )
        converts [min]


% Identifies the median in a set with odd number elements by declaring
% the number of elements smaller than the median and the number of
% elements greater than the median are equal

Median: trait
        imports RangeCount, Max, Min
        introduces median: C → E
        constrains median so that for all [ b: C]
            rangecount(b, min(b), median(b)) =
                rangecount(b, median(b), max(b))
        converts [median]


% A set with even number of elements has two candidates for the median.
% Median1 identifies the smaller one whereas Median2 identifies the
% larger one.

Median1: trait
```

```
        imports RangeCount, Max, Min
        introduces median1: C → E
        constrains median1 so that for all [ b: C]
            rangecount(b, min(b), median1(b)) + 1 =
                rangecount(b, median1(b), max(b))
        converts [median1]


Median2: trait
        imports RangeCount, Max, Min
        introduces median2: C → E
        constrains median2 so that for all [ b: C]
            rangecount(b, min(b), median2(b)) - 1 =
                rangecount(b, median2(b), max(b))
        converts [median2]

% Test if a set has odd number elements

Odd: trait
        imports AdditiveSize
        introduces odd: B → Boolean
        constrains odd so that for all [b:C]
            ( odd(b) ) = ( size(b) = 2 * n + 1 )
        converts [odd]


% Identifies the element which has the maximum value all the elements
% smaller than median1

LeftM1: trait
        imports Min, Median1, RangeCount
        introduces leftm: C → E
        constrains leftm so that for all [ b:C]
            rangecount(b, min(b), leftm(b)) + 1
                = rangecount(b, min(b), median1(b))
        converts [leftm]


% Identifies the element which has the minimum value all the elements
% larger than median1

RightM2: trait
        imports Max, Median2, RangeCount
```

```
        introduces rightm: C → E
        constrains rightm so that for all [ b:C]
            rangecount(b, rightm(b), max(b)) + 1
                = rangecount(b, median2(b), max(b))
        converts [leftm]


% Counts the number of valid elements (channels). An element will be
% counted if its value passes the threshold test

CountChannel: trait
        assumes Container, Cardinal
        introduces ccount: C, E, E, E → Integer
        constrains ccount so that for all [b:C, e,e1,e2,e3:E]
            ccount({}, e1, e2, e3) = 0
            ccount(insert(b,e), e1, e2, e3) =
                ccount(b, e1, e2, e3) + (if (abs(e1+e2+e3) > e &
                    abs(e1-e2-e3) < e) then 1 else 0)
        converts [ccount]
```

The following is the Larch/Pascal interface language part of the Real Number Decision Algorithm specification.

```
% Determine whether majority exists.
% Declares the median as the correct result if there is majority.
% Distinguishable median operators are utilized for different cases

type Rset exports Majority
        based on sort C from MultiSet with [real for E]
function Majority (b, sp, sn: Rset; var correctreal: real): Boolean
        modifies at most [correctreal]
        ensures
            if odd(b) then
                if rangecount(b,min(b),median(b))
                        ≤ ccount(b,median(b),median(sp),median(sn))
                    then Majority & correctreal = median(b)
                    else ¬Majority & modifies nothing
                else if median1(b)-leftm(b) < rightm(b)-median2(b)
                    then if rangecount(b,min(b),median1(b))
                                ≤ ccount(b,median1(b),median1(sp),median1(sn))
                            then Majority & correctreal = median1(b)
                            else ¬Majority & modifies nothing
                    else if rangecount(b,min(b),median2(b))-1
```

≤ ccount(b,median2(b),median2(sp),median2(sn))
then Majority & correctreal = median2(b)
else ¬Majority & **modifies nothing**

The traits concerning medians all include the library operator *rangecount*. The rangecount is defined in the trait RangeCount which counts the number of elements that fall into the range (e1, e2). Its quantified equation is

rangecount(insert(c,e_3),e_1,e_2) =
rangecount(c) + **if** ($e_3 \geq e_1$ & $e_3 \leq e_2$) **then** 1 **else** 0

Then the quantified equation for median (for a set with odd number of elements) is:

rangecount(b, min(b), median(b)) =
rangecount(b, median(b), max(b))

The motivation is to define a median value without sorting and indexing a sequence. Unfortunately, this definition fails on some multisets such as { 2, 2, 3, 3, 4 }. In that set, the median is 3, the minimum is 2, and the maximum is 4. The number of elements fall into the interval [min, median] is 4, but the number of elements fall into the interval [median, max] is 3. Thus the theorem stated by the above equation is not valid. The boundary conditions were overlooked.

5.2.2.2 Three Different Versions of the Median Trait

The First modified version of Median

Closer observation of a median value showed the fact that both the number of elements falling in the interval between the minimum and the median and the number of elements falling into the interval between the maximum and the median must be greater or equal to half the size of the multiset. Then a modified version of the median definition which still uses *rangecount* has been constructed as follows:

```
Median: trait
includes RangeCount, AdditiveSize, BagBasics,
        Pair with [P for C]
introduces
        csize: C → Integer
        cand: C → P
        median: C → E
constrains C so that for all [b:C]
        csize(b) = (size(b) + if odd(b) then 1 else 0)/2
        rangecount(b, cand(b).first, max(b)) ≥ csize(b)
        rangecount(b, min(b), cand(b).first) ≥ csize(b)
        rangecount(b, cand(b).second, max(b)) ≥ csize(b)
        rangecount(b, min(b), cand(b).second) ≥ csize(b)
        median(b) = if min(cand(b)) -
            min(cand(delete(delete(b,min(cand(b))),max(cand(b))))))
            ≤ max(cand(delete(delete(b,min(cand(b))),max(cand(b))))) - max(cand(b))
                    then min(cand(b))
                    else max(cand(b))
```

Since the multiset can have either an odd or an even number of elements, an intermediate operator *cand* is introduced, which always returns two candidates for the median in the sort *Pair*. Median is then determined by further numerical checking according to the particular definition of median in the Real Number Decision Algorithm. The advantage of this solution is the mathematical strictness. The disadvantage is poor comprehensibility. It is hard to build a clear picture of the median in the reader's mind at his first few glances. Furthermore, there is a subtle error, i.e., the operator *cand* might return a pair in which one element is a duplicate of the other so that the definition of median of a set with an even number of elements is violated.

The second modified version of the Median

A suggestion was made during the seminar class discussion. The approach suggested was to delete the extremas from the multiset successively. If the set has an odd number of elements, the deletion stops when there is only one left which is the

median. If the set has an even number of elements, the deletion stops when there are two elements left which are the median candidates. The new version is as follows.

```
Median: trait
includes AdditiveSize, BagBasics, Join, Pair with [P for C]
introduces
        oddmd: C → E
        evenmd: C → E
        evencand: C → P
        neighbors: C → P
constrains C so that for all [b:C]
        odd(b) => oddmd(b) = if size(b)=1 then b
                        else oddmd(delete(delete(b, min(b))), max(b))
        ¬odd(b) => evencand(b) = if size(b)=2 then b
                        else evencand(delete(delete(b, min(b))), max(b))
        & neighbors(b) = if (size(b)=2 | size(b)=4) then (min(b) .join max(b))
                        else neighbors(delete(delete(b, min(b))), max(b))
        & evenmd(b) = if min(evencand(b)) - min(neighbors(b))
                ≤ max(neighbors(b)) - max(evencand(b))
                then min(evencand(b))
                else max(evencand(b))
```

The advantage here is the intuitiveness; the disadvantage is the implementation bias which otherwise could be avoided.

The third modified version of the Median

Another way to define a median is to divide a multiset into two multisets. The two multisets have the same size and every element in one set is greater than or equal to any element in the other set. If the original set has an even number of elements, the two new sets are disjoint. If the original set has an odd number of elements, the two new sets will have an intersection. The intersection, however, cannot be any value other than the median since it is asserted that each element in one set will be greater than or equal to any element in the other set. Thus, the maximum of the smaller set and the minimum of the larger set are the candidates for the median. The trait is as

follows:

Median: **trait**
includes AdditiveSize, Join, SetIntersection, BagBasics
introduces
isSpan: C, C, C → Bool
median: C → E
constrains C **so that for all** [b:C]
isSpan(b,b_1,b_2) = (b = **if** odd(b) **then** delete(b_1 .join b_2, $b_1 \cap b_2$)
else b_1 .join b_2) &
size(b_1) = size(b_2) &
max(b_1) ≤ min(b_2)
isSpan(b, b_1, b_2) => median(b) =
if max(b_1) - max(delete(b_1, max(b_1)))
≤ min(delete(b_2,min(b_2))) - min(b_2)
then max(b_1) **else** min(b_2)

This approach retains the mathematical rigor, and its assertive nature gives a high level definition. It is also fairly easy to understand. However, this technique uses an implicit definition in terms of axioms rather than an implicit definition in terms of algebraic relations. All three versions do not exhibit the preferred style of Larch and lose harmony with other parts of the specifications.

5.2.3 Discussions with Drs. J.V. Guttag and J.J. Horning

Drs. Guttag and Horning, the Larch specification language designers, wrote a Larch specification of Real Number Decision Algorithm, which became the most valuable material for learning the language [Gutt86]. The following is his specification:

```
Nth: TRAIT
    ASSUMES Container, Size, Ordered WITH [E for T]...
    IMPORTS Cardinal
    INTRODUCES
      nth: C, Card -> E
      median: C -> E
      funnyMedian: C, Card -> E
```

```
    belowMedian: C -> Card
    countBelow: C, E -> Card
    withinSkew: C, E, E -> Card
  CONSTRAINS C SO THAT FOR ALL [n: Card, c: C, e, e', skew: E]
    nth(insert(c, e), n) = IF isEmpty(c) THEN e
        ELSE IF e >= nth(c, n) THEN nth(c, n)
        ELSE max(e, nth(c, n-1))

    median(c) = IF odd(size(c)) THEN nth(c, (size(c)+1)/2)
        ELSE funnyMedian(c, size(c)/2)
    funnyMedian(c, n) =
        IF (nth(c, n) - nth(c, n-1)) <= (nth(c, n+2) - nth(c, n+1))
        THEN nth(c, n) ELSE nth(c, n+1)

    belowMedian(c) = countBelow(c, median(c))
    countBelow(new, e) = 0
    countBelow(insert(c, e'), e) =
        countBelow(c, e) + (IF e' < e THEN 1 ELSE 0)

    withinSkew(new, e, skew) = 0
    withinSkew(insert(c, e'), e, skew) =
        withinSkew(c, e, skew) +
        (IF (e' >= (e-skew)) & (e' <= (e+skew)) THEN 1 ELSE 0)
  EXEMPTS FOR ALL [n: Card]
    nth(new, n)
    belowMedian(new)

FUNCTION Majority(b, sp, sn: vec; var correctreal: e): Boolean
  MODIFIES AT MOST [correctreal]
  ENSURES
    IF (belowMedian(b)
        <= withinSkew(b, median(b), median(sp)+median(sn)))
            THEN Majority & correctreal!post = median(b)
            ELSE ~Majority & MODIFIES NOTHING
```

The first noticeable feature of Drs. Guttag and Horning's specification is that all the behavior is defined by the algebraic relationships between the arguments of the operations and the results. Different behaviors are described in different quantified equations in a quite consistent manner. This minimizes the distraction of readers and improves comprehensibility.

Second, a larger trait introduces all the operators except some reusable operators such as max and odd, which would be introduced in some smaller traits

individually. This approach reduces the size of the specification tremendously because one additional trait will give additional heading lines due to the requirement of the formality. Smaller size usually add lucidity.

Third, an operator *nth* is introduced. The powerful *nth* enables access to the nth largest element from a sequence. Its quantified equation implies the embedded indexing and embedded ordering of a sequence of elements. By introducing the operator *nth*, the axioms associated with the operator Median become straightforward.

Fourth, the operation to pick the median for these sets with an even number of elements now resides in the Shared Language part. The size of the interface language part has been minimized, and so has the implementation bias.

Nevertheless, two operators, *belowMedian* and *nth* are incorrectly defined. They have the undesired behavior discussed below.

5.2.3.1 The Operator belowMedian

The operator *belowMedian* counts the number of elements whose values are less than the median. It is used by the Larch/Pascal function Majority to decide if a majority exists.

```
IF belowMedian(b) <= withinSkew(b, median(b), median(sp)+median(sn))
      THEN Majority & correctreal!post = median(b)
      ELSE ~Majority & MODIFIES NOTHING
```

For example, a set of numbers {1, 1, 5, 5, 90, 100, 120} has a median value 5. If both median(sp) and median(sn) equal to 1, then the skew interval will be [4, 6]. The set will have only two elements, namely 5 and 5, within the skew interval. Consequently, no majority exists. However, the operator *belowMedian* gives a result of 2. Thus the function Majority claims the existence of a majority.

In order to make the specification work, the belowMedian could be replaced by an operator as

```
morethanHalf(b) = (size(b) + if odd(b) then 1 else 2) / 2
```

The first line underneath ENSURES could then be changed to

```
IF morethanHalf(b) <=
        withinSkew(b,median(b),median(sp)+median(sn))
```

5.2.3.2 The Operator nth

The *nth* operator is intended to define an implied ordered sequence. Its rational is as follows: If an element e is added to a multiset c, in the case this element is greater than the kth in c, the first k elements remain in the original order so that the kth in the set still remains as the kth in the new set. Otherwise the new element will be inserted into somewhere between the first and the kth element and then either the new element e or the (k-1)th element in c becomes the new kth element depending on their ordering. On the two boundaries,

1. Accessing the very first element, nth(c, 1).
2. Accessing the very last element, nth(c, size(c)).

nth fails on certain permutations of the substitution (order of resolution). For a set {1, 2}, nth(insert({1}, 2), 1) returns 1 but nth(insert({2}, 1), 1) returns 2. Similarly, nth(insert({2}, 1), 2) returns 2 but nth(insert({1}, 2), 2) returns 1.

The first tentative revised version is:

```
nth(insert(c,e),n) = if (n = size(c) + 1)
                     then max(nth(c,n-1), e)
```

else if e ≥ nth(c,n) **then** nth(c,n)
else max(nth(c,n-1),e)

% nth(c,0) = neginfinity
% 0 ≤ n ≤ size(c)

where the condition "IF n = size(c) + 1" assures the retrieval of the very last element and the definition "nth(c, 0) = neginfinity" assures the retrieval of the very first element. Thus *nth* survives these boundary cases.

In order to get rid of "neginfinity," the second revised version was done as follows.

constrains C **so that for all** [k, n:card, c:C, e: E]
nth(c,0) ≤ e
nth(insert(c,e),n) = **if** (n = size(c) + 1)
then max(nth(c,n-1), e)
else if e ≥ nth(c,n) **then** nth(c,n)
else max(nth(c,n-1),e)
exempts for all [n:card, c:C, n > size(c)] nth(c,n)

The final version sent to Drs. Guttag and Horning is:

constrains C **so that for all** [k, n:card, c:C, e: E]
nth(c,0) ≤ e % nth(c,0) gives the "absolutely" smallest element
nth(insert(c,e),k+1) = **if** k = size(c)
then max(nth(c,k), e)
else if e ≥ nth(c,k+1) **then** nth(c,k+1)
else max(nth(c,k),e)
exempts for all [n:card, c:C, n > size(c)] nth(c,n)

% nth(new,n) for n > 0 is exempted because n > size(new)
% nth(new,0) will have nth(c,0) ≤ e for all e:E
% so nth(new,n) need not to be exempted here

Dr. Horning gave the following comments [Horn86]:

```
It took me a while to grok this one, but it seems to work, except for
the EXEMPTS. I would have used a >= in the first conditional.
```

The first inequality startled me. I guess it's a perfectly good way to avoid naming the least element of the type, but I'd normally give it a name, and turn this into an equality.

I would tend to use k on the left hand side, and k-1 on the right, but that is a small matter of style.

Another solution suggested by Jean-Paul Blanquart is:

```
CONSTRAINS nth SO THAT
     FOR ALL [c:C, size(c)>0, e:E, p:Integer, 0<p<size(c)+1]
nth(insert(new,e),1) = e
nth(insert(c,e),size(c)+1) = max(e,nth(c,size(c)))
nth(insert(c,e),1) = min(e,nth(c,1))
nth(insert(c,e),p) =
          if e >= nth(c,p)  then  nth(c,p)
          .                 else  max(e,nth(c,p-1))
IMPLIES CONVERTS(nth)
EXEMPTS nth(new,p) FOR all p,
          nth(c,p) FOR ( p <= 0 OR p > size(c) )
```

Dr. Horning's comments are as follows [Horn86]:

Over the years we have found that it is usually best to write (almost all) left-hand sides of equations in a standard form: an "observer" function (such as nth), applied to variables and/or "constructor" functions (such as insert). Among other things, if an observer is applied to all the operators in a GENERATED BY, and the right-hand sides are simplifications (in a certain technical sense) then the observer is converted. More complex discrimination on the values of arguments we tend to put in conditionals on the right hand side.

Thus, we would tend not to write left hand sides like
     nth(insert(c,e),size(c)+1)

Also, your first and second (and third and fourth) equations have overlapping left hand sides. This is a terribly easy way to generate inconsistencies. Again, we tend to put a conditional on the right.

Finally, EXEMPTS cannot be conditional; it must be on the syntactic form of the term, not its value.

Meanwhile, Dr. Horning tried to fix *nth* in own way as follows [Horn86]:

```
Nth: TRAIT
 ...
```

```
    CONSTRAINS C SO THAT FOR ALL [n: Card, c: C, e, e', skew: E]
(s1)  nth(insert(c, e), n) = IF isEmpty(c) THEN e
(s2)       ELSE IF n > size(c) THEN max(e, nth(c, n))
(s3)       ELSE IF e >= nth(c, n) THEN nth(c, n)
(s4)       ELSE max(e, nth(c, n-1))
 ...
```

A number of cases that ware considered by Dr. Horning to convince himself that it captures the intuitive notion are as follows [Horn86]:

(c1) The first two cases ensure that *nth* returns the largest value in c when n is greater than its size.

(c2) If e is greater than or equal to the value of the nth element in the smaller set, it doesn't affect the value of the nth element in the larger set.

(c3) If e is between the value of the nth and (n-1)th, it is the nth value of the larger set.

(c4) If e is less than the (n-1)th, the (n-1)th becomes the nth.

Let's first look at an example which shows the incorrectness of this nth.

nth(insert({20},10),1)

→ s1: false

→ s2: false since not (1 > size({20}))

→ s3: IF 10 >= nth({20},1)

right hand side is nth(insert({},20),1) so when it recurses back, s1 becomes true and returns 20, which makes s3's condition false.

→ s4: max(10, nth({20},0))

the second argument of max is nth(insert({},20),0), which satisfies the condition of s1 and returns 20. Finally we get the result from max(10,20), which is 20. But the answer should be 10!

Where does this *nth* fail? We may check the behavior of this trait:

s1 and s2 handle the c1 as desired. s3 handles the c2 without problem. And s4 attempts to handle c3 and c4, however, s4 fails in the case which has n equal to 1 (so there is no (n-1)th element). In that case, the s4 makes an undesired recursion to s1 and returns the only element. The only element is certainly larger than the other operand of *max* because of the condition checking in s3. Thus this *nth* fails.

If we want to keep the original s1..s3, then the correction could be done by changing the s4 as follows.

```
ELSE IF (n=1) THEN min(e, nth(c,1))
ELSE max(e, nth(c, n-1))
```

5.3 Summary of Experience

The experiences gained in the learning process of Larch is valuable for specifying RSDIMU.

5.3.1 An Efficient Way to Learn Larch

Compared with the task to specify the practical application RSDIMU, the Median trait is so little and the *nth* operator is so tiny. However, these little things have been provided a great opportunity to learn Larch.

To write a troublesome specification in many alternative versions is indeed a effective way to learn Larch. One piece of advice from Dr. Horning is as follows [Horn86]:

```
I would encourage you to try writing a troublesome specification in
as many different styles as you can think of -- change the way the
```

data is grouped, the allocation of functions to traits, etc. This is a stage that is hard to teach; often we spend a day or more just moving equations around, renaming functions, etc. Two people at a whiteboard seem to be able to do this better than one, although often one will come back the next morning with a still better idea that occurred overnight.

On the way to explore some better specifications among all the alternatives, one can get familiar with a broad range of Larch syntax and semantics, and obtain a good sense on the spirit of this specification language.

5.3.2 The Spirit of Larch Style

Although many alternatives of Median have been written, the most acceptable style would be the latest version of the *nth* (see Sec. 5.2.3.2).

```
Nth: TRAIT
 ...
    CONSTRAINS C SO THAT FOR ALL [n: Card, c: C, e, e', skew: E]
      nth(insert(c, e), n) = IF isEmpty(c) THEN e
          ELSE IF n > size(c) THEN max(e, nth(c, n))
          ELSE IF e >= nth(c, n) THEN nth(c, n)
          ELSE IF (n=1) THEN min(e, nth(c,1))
          ELSE max(e, nth(c, n-1))
 ...
```

From specifying *nth*, we can have some insight into the spirit of Larch.

1. The algebraic Larch specification technique is based on a generalization of the algebraic construction known as a presentation.

2. The set of legal expressions is not defined by separate axioms, but by the convention that it is the set of expressions that can be formed from the operations given for the data abstraction such that type correctness is preserved.

3. The definitions for most data abstractions require the use of conditional

equations, that is, equations which do not hold for all possible substitutions of expressions for variables, but which hold only for substitutions which satisfy some condition.

4. In a good Larch specification, most of the programming-language-independent complexity is pushed into the traits, allowing interface specifications to become almost trivial.

The spirit of Larch has to be in the specifier's mind throughout the process of specifying a problem. All the specifications must be in harmony in their styles. Consistency on styles among all the traits is essential. This consistency will strengthen the comprehensibility of a specification because the formats that the programmer needs to follow are standard. This consistency will diminish implementation bias because it enables programmers to distinguish the expression of functional requirements and the implied algorithm.

5.3.3 Simplicity vs. Complexity in Writing Specifications

The syntax of Larch is comparatively simple. The Larch Shared Language is perhaps more notable for what it leaves out than for what it includes. Some features found in other algebraic specification languages have been omitted in Larch. The gain in simplicity of language is at the cost in expressive power or increase in the complexity of specifications written in the language. However, the number of approaches devised for specifying a median shows the power of the simple syntax. After one perceives the spirit of Larch, he can convince himself that Larch is able to capture most properties of most problems.

5.3.4 To Uncover Specification Errors

The experience on Median and *nth* suggests that the process of writing specifications is at least as error-prone as the process of programming. The ultimate tool for error detection is the understanding of human minds.

While discussing the Larch specifications, several errors were uncovered by inspection; errors in Larch are easier to detect and ambiguities are more likely to be recognized than those in English. The reason is that the algebraic specification is understood in a formal way.

The experience suggests the necessity of a symbolic execution tool accompanying the automated theorem prover. A theorem prover is only able to uncover inconsistencies in the specifications. Therefore, as long as there is no inconsistency, mismatching of the effort of a specification and the functional requirements intended to be specified would not be uncovered by the theorem prover. For example, the serious errors that occurred in the operator *nth* will never be uncovered by a theorem prover but may be exposed by the output of a symbolic execution tool which is able to trigger the boundary conditions.

CHAPTER 6

THE LARCH SPECIFICATION FOR RSDIMU

6.1 General Structure of the Larch Specification

The functional requirements of RSDIMU can be divided into four modules. They are:

- *Calibration*
- *Fault Detection*
- *Estimation*
- *Display*

The System Data Flow Diagram is shown in Figure 1.

Study of the existing English specification shows that most deficiencies are in the first three modules. Functional requirements in Display are relatively straightforward. The effort of that part of the specification is rather satisfactory. The figures, tables and descriptions in that module are clear. The same intuitiveness may not be achieved by the formal specification. So we have come to the decision to leave that module in English. In addition, the English specifications are interspersed with environmental details, e.g., version restrictions, the version of the system that maintains it, write permission in the release directory, etc. These are important, and they should be specified separately from the Larch specifications part. Formal specifications cannot entirely replace informal specifications -- they are

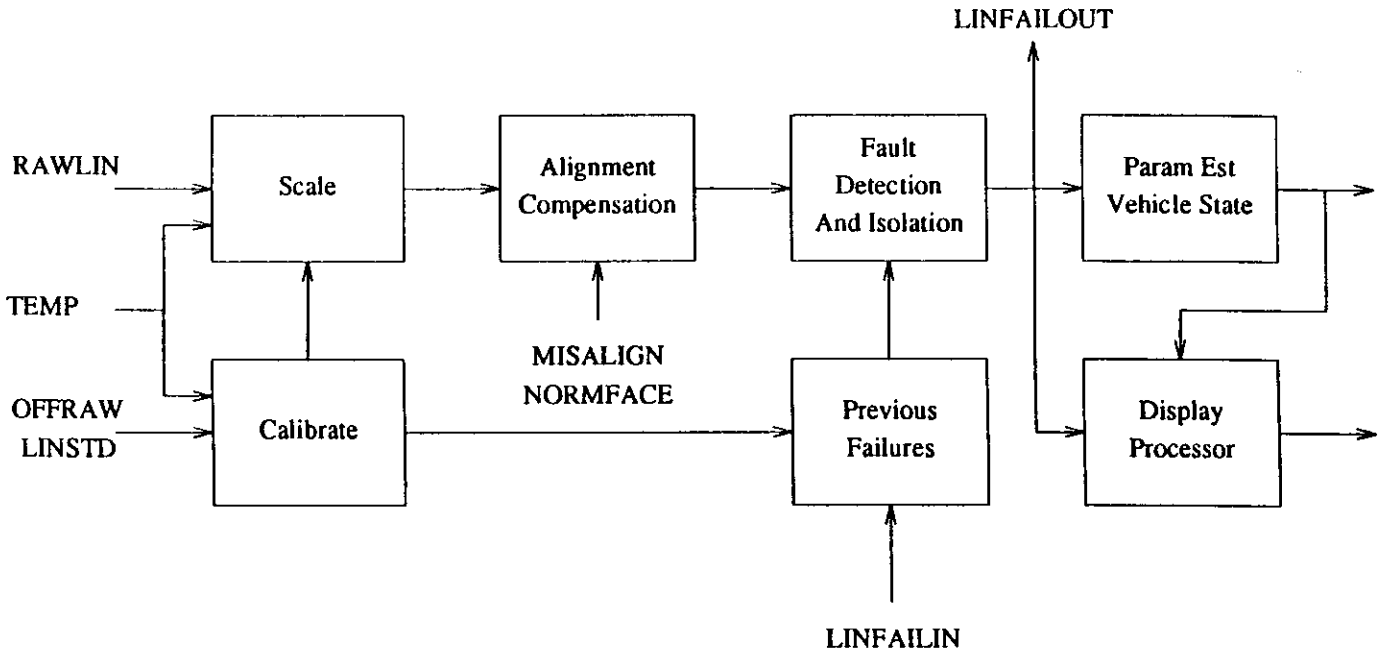

Figure 1. RSDIMU System Data Flow Diagram

complementary. Ideally, system specifications should include both formal and informal specifications. The informal specifications are easier to read and understand while the formal specifications tend to be clearer, precise, unambiguous, etc. Whenever there are any doubts about the informal specifications, the formal specifications should be used to resolve doubts [Geha82].

I do not yet have experience with the use of Larch specification in practical software development, and the supporting tools are not yet available. However, the promise of Larch for dealing with practical problems is being assessed, and I am encouraged with the way the most of functional requirements in RSDIMU seem to be captured by Larch.

The Larch specifications for the three modules of RSDIMU are 750 lines long. Its local structure diagrams are shown on figures 2, 3, and 4. Other details are shown in Table 2. The complete specifications is presented in the Appendix.

Modules	Traits	L/P Procedures	Lines	Resuable Traits
Calibration	18	4	267	10
Fault-Detect	15	1	190	7
Estimation	10	2	293	1
Total	43	7	750	14

Table 2: Sizes of RSDIMU Larch Specifications

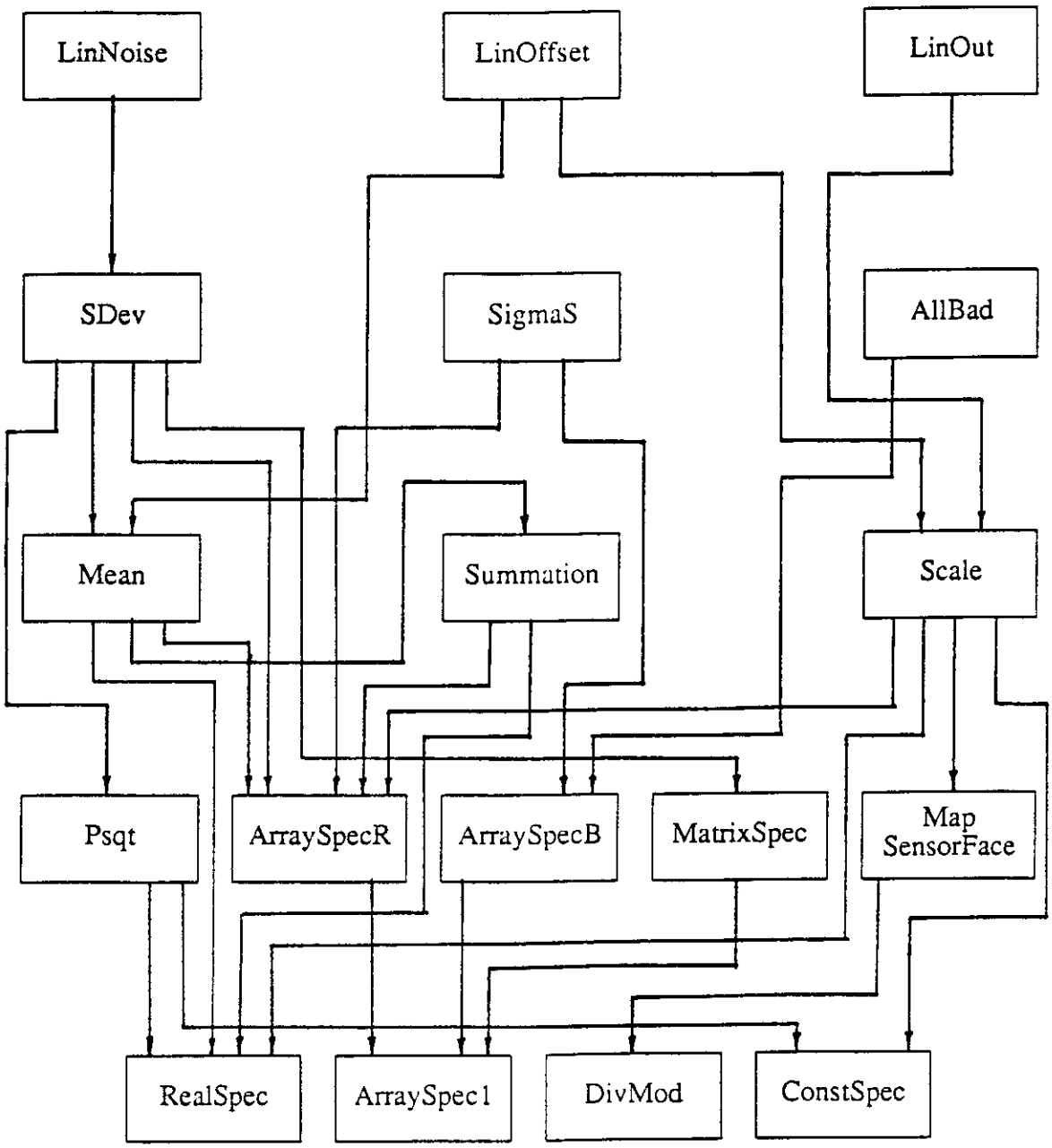

Fig. 2: Relations Among the Traits for the Calibration Module

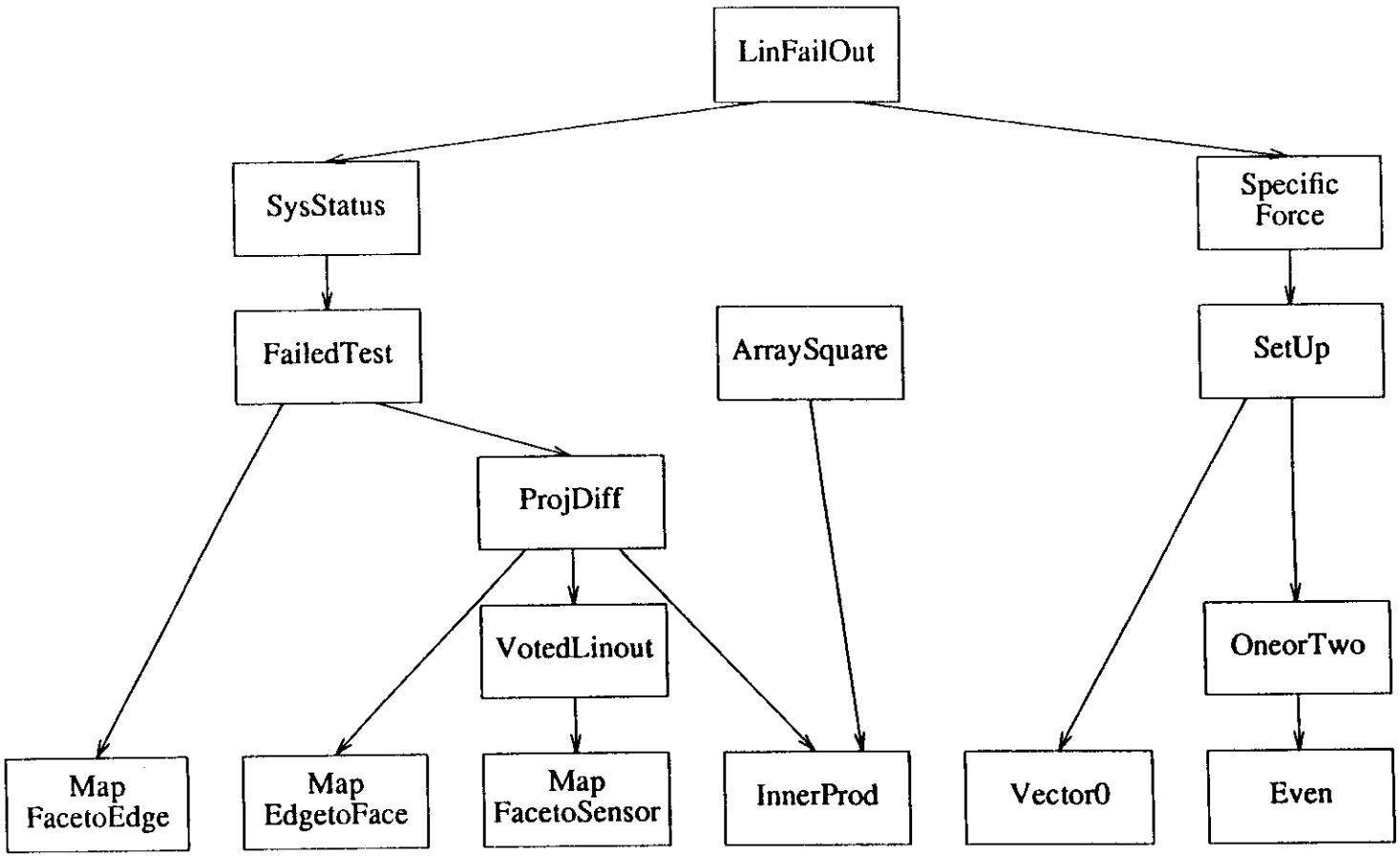

Fig. 3: Relations Among the Traits for the Fault-Detection Module

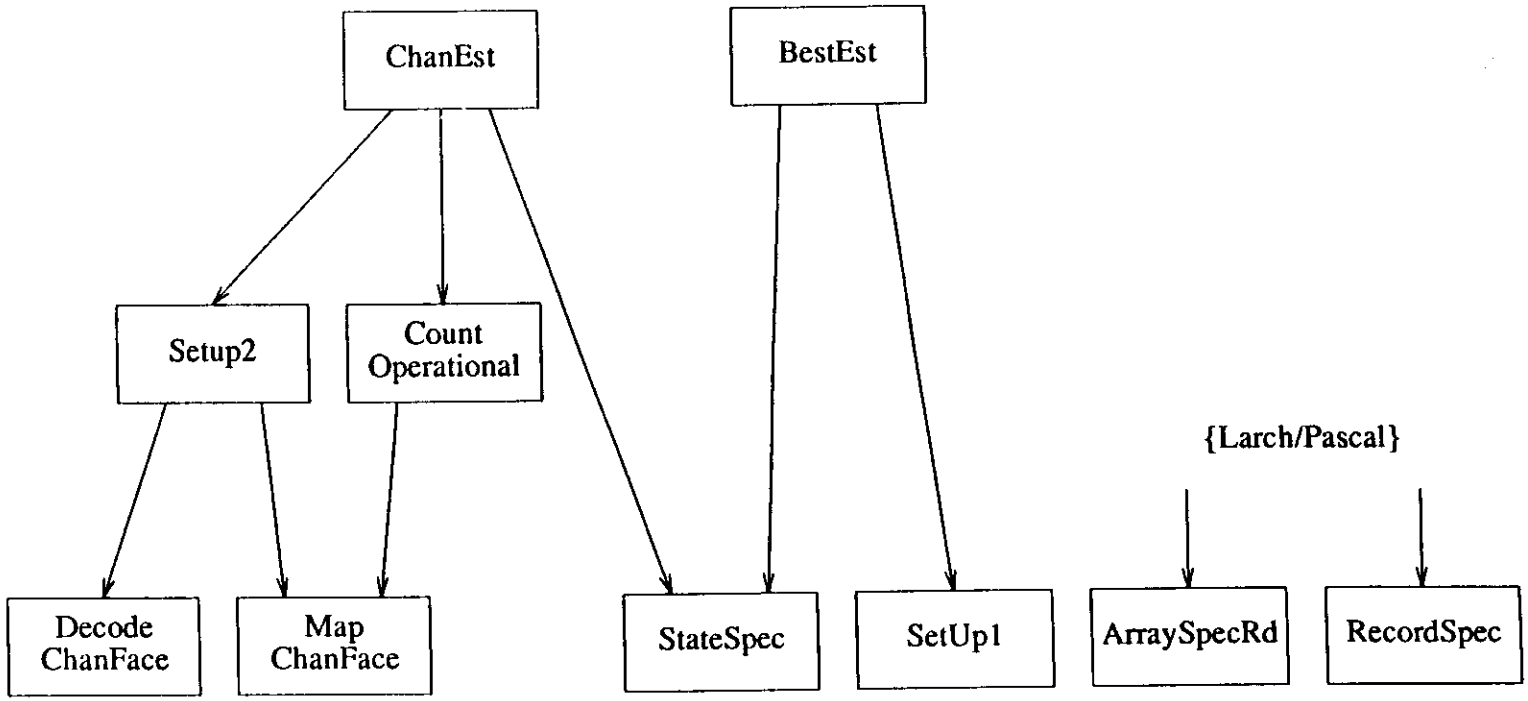

Fig. 4: Relations Among the Traits for the Estimation Module

Because of the reusability and composability of Larch, the rate of growth of the size of the specification decreases, while the amount of information captured by the specifications increases. In other words, the growth of size of the specification will not be linear.

6.2 Using Larch to Specify RSDIMU

RSDIMU is a medium size program. The average size of the 20 programs is 2000 lines. The specifications involve data abstractions, mathematical fundamentals, system definitions, basic algorithms, and special requirements of fault-tolerant software.

6.2.1 Data Abstractions

Data abstractions in RSDIMU include array which is utilized from input of raw data to final report of the health of the sensors, matrix which used for geometry transformation, real number sort, etc. All of them are specified in an incremental way.

Array type is specified in the library trait ArraySpec. In order to carry out some mathematical functions such as summation and inner product, the original ArraySpec with its incomplete nature is not sufficient. For example, definitions of upper bound and lower bound are required. The ArraySpec1 gives an increment to ArraySpec by importing ArraySpec and adding two conventional quantified equations in recurrence fashion.

```
ArraySpec1: TRAIT % increment of ArraySpec
IMPORTS ArraySpec
INTRODUCES upper, lower: Array -> Integer
```

```
CONSTRAINS Array SO THAT FOR ALL [t:Array, ind: Integer, val: Val]
      lower(assign(t, ind, val)) = IF isEmpty(t) THEN ind ELSE
            IF ind < lower(t) THEN ind ELSE lower(t)
      upper(assign(t, ind, val)) = IF isEmpty(t) THEN ind ELSE
            IF ind > upper(t) THEN ind ELSE upper(t)
EXEMPTS lower(new), upper(new)
```

The first version of MatrixSpec simply uses the renaming mechanism:

ArrayofArraySpec: **trait**
includes ArraySpec **with** [vector **for** val, ArrayofArray **for** Array]

However, it presents some implementation bias. E.g., it implies a data structure "array of array" in Pascal and excludes the two dimensional array data structure. Also there would be no way to directly access an individual cell. So a new version is:

```
MatrixSpec: TRAIT
IMPORTS ArraySpecR, IntegerSpec
INTRODUCES
      new: -> Matrix
      assign: Matrix, Integer, Integer, Val -> Matrix
      defined:  Matrix, Integer, Integer -> Bool
      read: Matrix, Integer, Integer -> Val
      isEmpty: Matrix -> Bool
      row: Matrix, Integer -> Array
      column: Matrix, Integer -> Array
      elecount: Matrix -> Card
CONSTRAINS Matrix SO THAT
      Matrix GENERATED BY [new, assign]
      Matrix PARTITIONED BY [defined, read]
      FOR ALL [r, c, ind1, ind2: Integer, val: Val, m: Matrix]
      read(assign(m, ind1, ind2, val), r, c) = IF (ind1 = r) and
                        (ind2 = c) THEN val ELSE read(m, r, c)
      defined(new: -> Matrix, r, c) = false
      defined(assign(m, ind1, ind2, val), r, c) =
            ((ind1 = r) and (ind2 = c)) | defined(m, r, c)
      elecount(new: -> Matrix) = 0
      elecount(assign(m, r, c)) = IF defined(m, r, c) THEN elecount(m)
                  ELSE elecount(m) + 1
      isEmpty(m) = (elecount(m) = 0)
      read(row(m, r), c) = read(m, r, c)
      read(column(m, c), r) = read(m, r, c)
      %read(row(m, r), c) = read(column(m, c), r)
```

MatrixSpec now is actually an analog of ArraySpec with one more dimension.

Additional operators *row* and *column* extract one dimension arrays. This specification is favored by the composability of Larch.

An uncountable domain, such as the real numbers, cannot be defined using the algebraic approach (without using an uncountable of primitive constructors). The specifications need to deal with real numbers. The solution is to assume the theory of Rational Numbers and rename the R(ational) with Real. Thus, the reader will perceive the real number type number type whereas the theory behind is about Rational. Without the renaming technique, other specification languages such as Ina Jo, can only simulate a real number by scaling it to an integer. The elegance of the specification is then ruined.

6.2.2 Mathematical Fundamentals

Several mathematical fundamentals have been specified for RSDIMU.

6.2.2.1 Summation

The operator *sum* is defined conventionally in a recurrence equation:

```
Summation: trait
ASSUMES ArraySpecR, IntegerSpec, RealSpec
INTRODUCES sum: ArrayR, Integer, Integer -> Real
CONSTRAINS sum SO THAT FOR ALL [t: Array, i,k: Integer]
      sum(t, i, k) = read (t, i) + IF (i = k) THEN 0
                     ELSE sum(t, i+1, k)
```

Incompleteness is left intentionally. The version with more completeness is the following:

```
sum(t, i, k) = IF defined(t, i) THEN read(t, i) ELSE 0
                  + IF (i = k) THEN 0 ELSE sum(t, i+1, k)
```

Since we never expect that a summation is computed on a sequence with some element undefined, and since the above definition may confuse the programmer, the *sum* has been changed to the current version and its definition was intentionally left incompleted. Dr. Horning has introduced *sum* as an operator taking just one argument which is an array together with an auxiliary operator *sumFrom* [Horn86]:

```
Summation: TRAIT
ASSUMES ArraySpec1, Real, Integer
INTRODUCES sum: Array -> Real
      sumFrom: Array, Integer -> Real
CONSTRAINS sum, sumFrom SO THAT FOR ALL [t: Array, i: Integer]
      sum(t) = sumFrom(t, lower(t))
      sumFrom(t, i) = read(t, i) +
            (IF i = upper(t) THEN 0 ELSE sumFrom(t, i+1))
EXEMPTS FOR ALL [i: Integer]
      sum(new)
      sumFrom(new, i)
```

The one-operand operator looks more attractive than the multi-operand operator. However, the cost for doing so is loss of flexibility. I.e., the one-operand case will be forced to sum up the whole sequence whereas the 3-operands case is able to sum up any subsequence. There is some way to achieve both the flexibility and elegance, which is to introduce an additional operator *sumAll*:

```
sumAll(t) = sum(t, lower(t), upper(t))
```

Thus the *sumAll* can be used to sum up the whole sequence, while the *sum* can be used to sum up any subsequence.

6.2.2.2 Division and Modulo

The first version of DivMod is:

```
DivMod: TRAIT
ASSUMES Cardinal
INTRODUCES div, mod: Card, Card -> Card
```

```
CONSTRAINS div, mod SO THAT FOR ALL [p, d: Card, d <> 0]
      d * div(p, d) =< p
      d * (div(p, d) + 1) > p
      p = d * div(p, d) + mod(p, d)
IMPLIES FOR ALL [p, d: Card] mod(p, d) < d
EXEMPTS [FOR ALL p: Card] div(p, 0), mod(p, 0)
```

Dr. Horning had the following comments [Horn86]:

We tend to view the use of inequalities standing on their own as somewhat of a last resort. This may be because we tried for so long to get along with them entirely, and because they don't really work all that well in a rewrite-rule based theorem-prover. However, I doubt that I could give as elegant a definition of div and mod as this one without using them. Cute.

I tried to rewrite it by using equality only. The second version is the following:

```
DivMod: TRAIT
IMPORTS Cardinal
INTRODUCES div: Card, Card -> Card
         mod: Card, Card -> Card
CONSTRAINS div, mod SO THAT FOR ALL [p, d: Card]
      div(p, d) = IF (p < d) THEN 0
                  ELSE (div(p-d, d) + 1)
      mod(p, d) = IF (p < d) THEN p
                  ELSE mod(p-d, d)
EXEMPTS [FOR ALL p: Card] div(p, 0), mod(p, 0)
```

If the infix notation is used, the equations will look like

```
p div d = IF (p < d) THEN 0
            ELSE ((p-d) div d + 1)
p mod d = IF (p < d) THEN p
            ELSE ((p-d) mod d)
```

Although in this case the assertional style of the former approach is more appealing, the later approaches are more acceptable because the theorem-prover is going to work more effectively when given equations.

It is encouraged to obtain the mutual exclusion by setting the conditional on the right hand side of the quantified equation. Trying to achieve mutual exclusion by overlapping left hand sides of equations always create a risk to generate inconsistency.

Failure to consider the boundary cases will make a specification invalid. Typical examples are the misuse of *rangecount* for defining a median (see pages 26-29) and ignoring the access to the extremas in a sequence for *nth* operator (see pages 35, 36, 38 and 39).

The desired results can be achieved only if the relevant syntax is fully understood. The trait AllBad used to be written as

```
AllBad: TRAIT
ASSUMES ArraySpec1
INTRODUCES allBad: Array -> Bool
CONSTRAINS allBad SO THAT FOR ALL [T: Array, ind: Card]
      allBad(t) = (read(t, ind) = true)
```

It seems to be a direct translation of the mathematical language: An array T is said to be allBad if the individual cell T[i] equal to TRUE for all i: i ∈ Integer. Unfortunately, the syntax works out the other way around: An array T is said "allBad" if there exists an i such that T[i] = TRUE.

In order to achieve correctness, it is always be beneficial to ask oneself "Does the syntax utilized work exactly in the desired way?"

Novices often put the conditional in some place outside the equations. E.g., in the Summation trait, instead of putting the conditional into the quantified equation or intentionally leaving incompleteness, the conditional for bounds of index was mistakely placed in the CONSTRAINS part. Similarly, one of the proposed traits *nth*

(see pages 36-37) places several conditionals in the EXEMPTS part.

IfThenElse branch can only be explicitly defined in the equations but can never be implicitly implied elsewhere. To say this more precisely, the universal quantification is only allowed over sorts; a predicate is not allowed in a FOR ALL.

Sort mismatching is also a common mistake. With a syntax checker, the failure on sort-check can be easily uncovered. Since now the detection of syntax errors totally depends on inspections, it is rather difficult to have all the specifications syntactically correct.

6.4 Getting Better Comprehensibility

There have been a number of interesting phenomena observed during the process of specifying RSDIMU.

6.4.1 Formalism vs. Readability

A formal language usually gives precise and unambiguous specifications. A formal specification can claim to be "free of sin" after undergoing all checks. It appears that the comprehensibility may suffer because of the formal approach. There are two approaches to define a zero-vector.

```
Vector0: TRAIT
ASSUMES Summation WITH [vector FOR Array, Card FOR Val]
INTRODUCES vector0: -> vector
ASSERTS size(vector0) = (upper(vector0) - lower(vector0)) + 1
      % no hole
      sum(vector0, upper(vector0), lower(vector0)) = 0
      % every slot is zero value

% An alternative which is less formal but more readable
Vector0: TRAIT
ASSUMES ArraySpec1 WITH [vector FOR Array, Card FOR Val]
INTRODUCES vector0: -> vector
```

```
CONSTRAINS vector0 SO THAT FOR ALL [ind: Card]
      read(vector0, ind) = 0
```

The first one looks elegant and sounds formal. However, programmers may favor the second one since the first one consumes their time trying to perceive its meaning.

An even more serious threat to the widespread use of algebraic specifications is the very considerable difficulty experienced by average programmers in understanding them, and even experts in their use make uncomfortably many mistakes while writing algebraic specifications. Resolution of the conflict between formality on one side and comprehensibility and constructibility on the other is an important research problem.

6.4.2 Tips to Improve Comprehensibility

Despite the conflict between formality and readability in algebraic specification languages, the better style always leads to the better comprehensibility. There are several instances:

1. To choose short names for sorts and to leave the longer names for traits. This makes an easy distinction between the names of sort and the names of traits.

2. Where multiple words are combined in an identifier, the convention needs to be uniformly used is that each word after the first is capitalized. This gives the reader a much stronger clue how the identifier is intended to be read. E.g., mapChanFace vs. mapchanface.

3. Another useful convention is to place the principal sort first in a signature, unless there is some strong reason for doing otherwise. E.g., it is preferable to have

```
mDefine: Matrix, Integer, Integer -> Bool
```

rather than

```
mDefine: Integer, Integer, Matrix -> Bool
```

4. Start sort identifiers with upper case letters and start operators with lower-case. This also provides an easier distinction.

5. Avoid "magic numbers" like 2048 or 409.6 in equations and introduce them as constants with meaningful names elsewhere in order to convey a little intuition.

```
ConstSpec: TRAIT
INCLUDES RealSpec, Equality WITH [Const FOR T]
INTRODUCES offsetCounts: -> Const
      scaleFactor: -> Const
CONSTRAINS Const SO THAT
      Const GENERATED By [offsetCounts, scaleFactor]
      FOR All []
                offsetCounts = 2048
                scaleFactor = 409.6
                BADDAT = 9999.0000
```

6. Always put the operator in the left hand side of an equation if it is not a recursive one.

6.5 Limitations of Larch

While we are pleased to see that Larch is able to capture most information of our application, we think that Larch specification technique requires further refinements.

Some additional features we wish Larch to have are:

1. *Good Shorthand Techniques.* When an operator has many data dependencies,

it must contain a long list of operands. This also affects comprehensibility. One way to solve the problem is to specify the group of operands as a record [Horn86].

```
Calibration RECORD OF [scale0, scale1, scale2: Array]
```

Thus the three operands can collapse to a single identifier Calibration. This is not considered a good way. First, this kind of shorthand implies a type declaration which otherwise would not be mentioned nor used. Second, in order to put actual operands to the fields of RECORD, assignments have to be done in the Larch/Pascal procedure such as

```
Calibration = valGets(Calibration, SCALE0, SCALE1, SCALE2)
```

which is another implementation bias.

Record notation is to be regarded purely as shorthand for a set of axioms, not as a hint to the implementor. However, in practice it is hard to say that this kind of shorthand provides no implementation bias. To use this kind of shorthand is not convenient because extra effort for assignments would be needed.

2. *Availability of Inequality.* Because inequalities do not really work all that well in a rewrite-rule based theorem-prover, the use of inequalities standing on their own is considered somewhat of a last resort [Horn86], as mentioned in section 5.2 for trait DivMod. In fact, inequality can contribute elegance and assertiveness, and reduce the amount of bias. Without inequality, many functions will become hard to define. Examples are:

   a. To use rational number to approximate a square root.

   b. To specify the least square method.

3. *Mixfix Notation.* Mixfix notations have not been available in Larch, and also Larch lacks the important and powerful predicate "there exist". We hope they would make Larch more powerful.

The ideas behind the Larch project are more important than its details [Horn85]. The main contribution of the Larch project lies in the understanding of the specification process and in their influence on the design of future specification methods that will be kinder to relatively unskilled users [Mell79].

Experience shows that the research oriented specification language Larch can be applied to practical problems in general. However, the range of the applicability of Larch is smaller than we would like. Since the range of applicability is different for the algebraic specification language and English, we may expect that using a combination of techniques when describing a large program such as the fault-tolerant software RSDIMU would be a profitable approach.

Even if we are not able to describe an entire program using formal specification techniques, the ability to define most of the modules used in constructing a system in a precise, formal way would be a major advance in the construction of reliable software. Work in applying existing formal specification techniques to practical problems, and in assessing the promise of the techniques, is of the utmost importance.

CHAPTER 7

CONCLUSION

The informal specification of a proposed software product is the most critical aspect of software production. Informal specifications, while easy to read, tend to be ambiguous, incomplete, imprecise and overspecific. Formal specifications offer the potential for the development of automated aids for detection of the above problems. They are valuable aids to fault-avoidance and fault-removal in the development of reliable software, and especially for Multi-Version Software systems.

The formal specification language Larch basically satisfies the general requirements demanded by fault-tolerant software, such as formality, constructibility, comprehensibility, minimality, extensibility [Kell83].

The merits of the algebraic specification technique, Larch, have been shown on fairly well studied examples such as stacks, queues, lists, etc. The transition from these data abstractions to a practical problem such as RSDIMU has been basically achieved after becoming acquainted with the spirit of Larch. A preliminary version of the Larch specification of RSDIMU has shown us that Larch is a promising formal specification language for fault-tolerant software, since it provides the accuracy of system definitions and the clarity of given algorithms.

Despite the conflict between formality and readability, better styles always lead to greater comprehensibility. Experience from the process of constructing the specification of RSDIMU suggests some additional features which may make Larch

more powerful, such as useful shorthands and the availability of inequality in the theorem prover. Since formal specification techniques have certain inherent limitations, such as limitations in comprehensibility and constructability in certain domains, we may expect that using a combination of techniques when describing a large program would be a profitable approach. The RSDIMU Larch specification is still immature and needs to be debugged and improved. Once the specification has been completed, it will be used by the programmers of the next generation multi-version software experiments.

REFERENCES

[Aviz77] A. Avižienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution," in *Proceedings COMPSAC 77,* 1977, pp. 149-155.

[Aviz82] A. Avižienis, "Design Diversity - The Challenge for the Eighties," in *Digest of 12th Annual International Symposium on Fault-Tolerant Computing,* Santa Monica, California: June 1982, pp. 44-45.

[Aviz85a] A. Avižienis, P. Gunningberg, J.P.J. Kelly, R.T. Lyu, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, "Software Fault-Tolerance by Design Diversity; DEDIX: A Tool for Experiments," in *Proceedings IFAC Workshop SAFECOMP'85,* Como, Italy: October 1985, pp. 173-178.

[Aviz85b] A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 12, December 1985, pp. 1491-1501.

[Boeh81] B. Boehm, *Software Engineering Economics*, Englewood, N.J.: Prentice Hall, 1981.

[Burs81] R.M. Burstall and J.A. Goguen, "An Informal Introduction to Specifications Using CLEAR," in *The Correctness Problem in Computer Science,* R. Boyer and H. Moore, Ed. New York: Academic Press, 1981, pp. 185-213.

[Redu85] CRA, "Redundancy Management Software Requirements Specification for a Redundant Strapped Down Inertia Measurement Unit," Charles River Analytics and Research Triangle Institute, Tech. Rep. Version 2.0, May 30, 1985.

[Futa85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," *Principles of Programming Languages, ACM,* Vol. 12, 1985, pp. 52-66.

[Geha82] N.H. Gehani, "Specifications: Formal and Informal - A Case Study," *Software - Practice and Experience,* No. 12, 1982, pp. 433-444.

[Gutt80] J.V. Guttag and J.J. Horning, "Formal Specification as a Design Tool," in *Proceedings 7th ACM Symposium on Principles of Programming Languages*, January 1980, pp. 251-261.

[Gutt83] J.V. Guttag and J.J. Horning, "An Introduction to the Larch Shared Language," in *Proceedings IFIP Congress 83*, 1983, pp. 809-814.

[Gutt85a] J.V. Guttag, J.J. Horning, and J.M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, Vol. 2, No. 4, September 1985, pp. 24-36.

[Gutt85b] J.V. Guttag, J.J. Horning, and J.M. Wing, "Larch in Five Easy Pieces," Digital Equipment Corporation Systems Research Center, Palo Alto, California, Tech. Rep. Report No. 5, July 24, 1985.

[Gutt86] J.V. Guttag and J.J. Horning, *Private communication*, 1986.

[Hehn84] E. Hehner, "Predicative Programming, Parts I and II," *Communications of the ACM*, Vol. 27, No. 2, February 1984, pp. 134-151.

[Horn85] J.J. Horning, "Combining Algebraic and Predicative Specifications in Larch," in *Proceedings Joint Conference on Theory and Practice of Software Development*, Berlin, Germany: March 1985.

[Horn86] J.J. Horning, *Private communication*, 1986.

[Kell82] J.P.J. Kelly, "Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach," UCLA, Computer Science Department, Los Angeles, California, Tech. Rep. CSD-820927, September 1982.

[Kell83] J.P.J. Kelly and A. Avižienis, "A Specification Oriented Multi-Version Software Experiment," in *Digest of 13th Annual International Symposium on Fault-Tolerant Computing*, Milan, Italy: June 1983, pp. 121-126.

[Kell86] J.P.J. Kelly, A. Avižienis, B.T. Ulery, B.J. Swain, R.T. Lyu, A.T. Tai, and K.S. Tso, "Multi-Version Software Development: UCLA's Perspective of a Second Generation Experiment," in *Proceedings IFAC Workshop SAFECOMP'86*, Sarlat, France: October 1986.

[Lisk77] B. Liskov and S. Zilles, "An Introduction to Formal Specifications of Data Abstractions," in *Current Trends in Programming Methodology, Volume 1*, R.T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 1-32.

[Lisk79] B.H. Liskov and V. Berzins, "An Appraisal of Program Specifications," in *Research Directions in Software Technology,* P. Wegner, Ed. Cambridge, MA: MIT Press, 1979, pp. 276-301.

[Loca80] R. Locasso, J. Scheid, V. Schorre, and P. Eggert, "The Ina Jo Specification Language Reference Manual," System Development Corp., Santa Monica, CA, Tech. Rep. TM-6889/000/01, November 1980.

[Mell79] P.M. Melliar-Smith, "System Specifications," in *Computing Systems Reliability,* T. Anderson and B. Randell, Ed. New York, NY: Cambridge University Press, 1979, pp. 19-65.

[Meye84] B. Meyer, "A System Description Method," in *International Workshop on Models and Languages for Software Specification and Design,* B.G. Babb II A. Mili, Ed. Orlando, Fla.: March 1984, pp. 42-46.

[Meye85] B. Meyer, "On Formalism in Specifications," *IEEE Software,* January 1985, pp. 6-26.

[Parn79] D.L. Parnas, "The Role of Program Specification," in *Research Directions in Software Technology,* P. Wegner, Ed. Cambridge, MA: MIT Press, 1979, pp. 364-370.

[Wing86] J.M. Wing, *Private communication,* 1986.

APPENDIX

```
% $Header: calibration,v 1.8 86/10/07 19:45:57 tai Exp $
```

% CALIBRATION MODULE

```
RealSpec: TRAIT
IMPORTS Rational WITH [Real FOR R]

ArraySpec1: TRAIT   % increment of ArraySpec
INCLUDES ArraySpec
INTRODUCES upper, lower: Array -> Integer
CONSTRAINS Array SO THAT FOR ALL [t:Array, ind: Integer, val: Val]
       lower(assign(t, ind, val)) = IF isEmpty(t) THEN ind ELSE
                                    IF ind < lower(t) THEN ind ELSE lower(t)
       upper(assign(t, ind, val)) = IF isEmpty(t) THEN ind ELSE
                                    IF ind > upper(t) THEN ind ELSE upper(t)
EXEMPTS lower(new), upper(new)

ArraySpecR: TRAIT
IMPORTS ArraySpec1 WITH [Real FOR Val, ArrayR FOR Array]

ArraySpecB: TRAIT
IMPORTS ArraySpec1 WITH [Bool FOR Val, ArrayB FOR Array]

MatrixSpec: TRAIT
IMPORTS ArraySpec1, IntegerSpec
INTRODUCES
       new: -> Matrix
       assign: Matrix, Integer, Integer, Val -> Matrix
       defined:  Matrix, Integer, Integer -> Bool
       read: Matrix, Integer, Integer -> Val
       isEmpty: Matrix -> Bool
       row: Matrix, Integer -> Array
       column: Matrix, Integer -> Array
       elecount: Matrix -> Card
CONSTRAINS Matrix SO THAT
       Matrix GENERATED BY [new, assign]
       Matrix PARTITIONED BY [defined, read]
       FOR ALL [r, c, ind1, ind2: Integer, val: Val, m: Matrix]
       read(assign(m, ind1, ind2, val), r, c) = IF (ind1 = r) and
                          (ind2 = c) THEN val ELSE read(m, r, c)
       defined(new: -> Matrix, r, c) = false
       defined(assign(m, ind1, ind2, val), r, c) =
              ((ind1 = r) and (ind2 = c)) | defined(m, r, c)
       elecount(new: -> Matrix) = 0
       elecount(assign(m, r, c)) = IF defined(m, r, c) THEN elecount(m)
                    ELSE elecount(m) + 1
       isEmpty(m) = (elecount(m) = 0)
       read(row(m, r), c) = read(m, r, c)
       read(column(m, c), r) = read(m, r, c)
       %read(row(m, r), c) = read(column(m, c), r)

Summation: trait
ASSUMES ArraySpecR, IntegerSpec, RealSpec
INTRODUCES sum: ArrayR, Integer, Integer -> Real
```

```
      sumAll: ArrayR -> Real
CONSTRAINS sum SO THAT FOR ALL [t: Array, i,k: Integer]
      sum(t, i, k) = read (t, i) + IF (i = k) THEN 0
                     ELSE sum(t, i+1, k)
      sumAll(t) = sum(t, lower(t), upper(t))
EXEMPTS FOR ALL [i, k: Integer] sum(new, i, k), sumAll(new)

DivMod: TRAIT       % Version 1
ASSUMES Cardinal
INTRODUCES div, mod: Card, Card -> Card
CONSTRAINS div, mod SO THAT FOR ALL [p, d: Card]
      d * div(p, d) =< p
      d * (div(p, d) + 1) > p
      p = d * div(p, d) + mod(p, d)
IMPLIES FOR ALL [p, d: Card] mod(p, d) < d
EXEMPTS [FOR ALL p: Card] div(p, 0), mod(p, 0)

DivMod: TRAIT       % Version 2
IMPORTS Cardinal
INTRODUCES div: Card, Card -> Card
         mod: Card, Card -> Card
CONSTRAINS div, mod SO THAT FOR ALL [p, d: Card]
      p div d = IF (p < d) THEN 0
                           ELSE (((p-d) div d) + 1)
      p mod d = IF (p < d) THEN p
                           ELSE ((p-d) mod d)
EXEMPTS [FOR ALL p: Card] (p div 0), (p mod 0)

MapSensortoFace: TRAIT
ASSUMES Cardinal
INTRODUCES senFace: Card -> Card
CONSTRAINS senFace SO THAT
      senFace(1) = 1
      senFace(2) = 1
      senFace(3) = 2
      senFace(4) = 2
      senFace(5) = 3
      senFace(6) = 3
      senFace(7) = 4
      senFace(8) = 4

ConstSpec: TRAIT
INCLUDES Equality WITH [Const FOR T]
INTRODUCES offsetCounts: -> Const
      scaleFactor: -> Const
      delta: -> Const
CONSTRAINS Const SO THAT
      Const GENERATED By [offsetCounts, scaleFactor, delta, BADDAT]
      FOR All []
            offsetCounts = 2048
            scaleFactor = 409.6
            delta = 0.000000001
            BADDAT = 9999.0000

Scale: TRAIT
ASSUMES ArraySpecR, MapSensortoFace, RealSpec, ConstSpec
INTRODUCES toVoltage: Real -> Real
```

```
          slope: ArrayR, ArrayR, ArrayR, ArrayR -> ArrayR
CONSTRAINS toVoltage, slope SO THAT FOR ALL
              [i: Integer, r: Real, scale0, scale1, scale2, temp: ArrayR]
       % try to give meaningful names to corresponding Real input
         variable names (but not exact)
       toVoltage(r) = (r - offsetCounts) / scaleFactor
       read(slope(scale0, scale1, scale2, temp), i)
              = read(scale0, i) +
                     read(scale1, i) * read(temp(senface(i))) +
                     read(scale2, i) * square(read(temp(senface(i))))

Mean: TRAIT
ASSUMES ArraySpecR, Summation, RealSpec
INTRODUCES mean: ArrayR -> Real
CONSTRAINS mean SO THAT FOR ALL [t: Array]
       mean(t) * size(t) = sum(t, lower(t), upper(t))

LinOffset: TRAIT
ASSUMES Scale, Mean
INTRODUCES linOffset: Matrix, ArrayR, ArrayR,
                           ArrayR, ArrayR, ArrayR -> ArrayR
CONSTRAINS linOffset SO THAT FOR ALL [gm, scale0, scale1, scale2, temp:
              ArrayR, offraw: Matrix, i: Integer]
       read(linOffset(offraw, gm, scale0, scale1, scale2, temp), i) =
              read(gm, i) - read(slope(scale0, scale1, scale2, temp), i) *
                           toVoltage(mean(row(offraw, i))))


PSqrt: TRAIT
ASSUMES RealSpec, ConstSpec
INTRODUCES pSqrt: Real -> Real
CONSTRAINS pSqrt SO THAT FOR ALL [r: Real]
       pSqrt(r) * pSqrt(r) =< r + delta
       pSqrt(r) * pSqrt(r) >= r - delta
       pSqrt(r) >= 0

SDev: TRAIT
ASSUMES ArraySpecR, PSqrt, Mean, MatrixSpec WITH [Real FOR Val]
INTRODUCES offMean: ArrayR, Card -> Real
           deltaSq: ArrayR, Card -> ArrayR
           sDev: ArrayR, Card -> Real
CONSTRAINS offMean, deltaSq, sDev SO THAT FOR ALL
              [offraw: Matrix, r, i: Integer]
       offMean(offraw, r) = mean(row(offraw, r))
       read(deltaSq(offraw, r), i) =
              square(read(row(offraw, r), i) - offMean(offraw, r))
       sDev(offraw, r) = pSqrt(sum(deltaSq(offraw, r),
              lower(deltaSq(offraw, r)), upper(deltaSq(offraw, r)))/
                     size(deltaSq(offraw, r)))

LinNoise: TRAIT
ASSUMES SDev
INTRODUCES linNoise: ArrayR, Integer -> Bool
CONSTRAINS linNoise SO THAT FOR ALL [offraw: ArrayR,
                                    linstd: Card, r: Integer]
       read(linNoise(offraw, linstd), r)
              = (sDev(offraw, r) > 3 * linstd)
```

```
AllBad: TRAIT
ASSUMES ArraySpecB
INTRODUCES allBad: ArrayB -> Bool
CONSTRAINS allBad SO THAT FOR ALL [t: Array, ind: Card, v: Bool]
      allBad(assign(t, ind, v)) = IF isEmpty(t) THEN v
                                  ELSE (v & allBad(t))

SigmaS: TRAIT
ASSUMES ArraySpecR, ArraySpecB
INTRODUCES addGood: Array, ArrayB, ArrayB, Integer, Integer -> Real
           countGood: Array, ArrayB, ArrayB, Integer, Integer -> Card
           meanSlope: Array, ArrayB, ArrayB -> Real
         sigmaS: Array, ArrayB, ArrayB, Integer -> Real
CONSTRAINS addGood, countGood, meanSlope, sigmaS SO THAT FOR ALL
             [slopearray: Array, noise, linfailin: ArrayB,
                                  ind1, ind2: Integer]
           addGood(slopearray, noise, linfailin, ind1, ind2)
             = (IF ~read(noise, ind1) &
                   ~read(linfailin, ind2) THEN read(slopearray, ind1)
                         ELSE 0) + IF (ind1 = ind2) THEN 0 ELSE
                               addGood(slopearray, noise, linfailin,
                                     ind1+1, ind2)
           countGood(slopearray, noise, linfailin, ind1, ind2)
             = (IF ~read(noise, ind1) &
                   ~read(linfailin, ind2) THEN 1 ELSE 0) +
                         IF (ind1 = ind2) THEN 0 ELSE
                               countGood(slopearray, noise, linfailin,
                                     ind1+1, ind2)
           meanSlope(slopearray, noise, linfailin) =
             addGood(slopearray, noise, linfailin,
                   lower(linfailin), upper(linfailin)) /
             countGood(slopearray, noise, linfailin,
                   lower(linfailin), upper(linfailin))
         sigmaS(slopearray, noise, linfailin, linstd) =
             meanSlope(slopearray, noise, linfailin) * linstd / scaleFactor
      % Here I assume that slopearray = slope(slope0, slope1, slope2, temp)
      % Does Larch allow us to do this kind of abbreviations?
EXEMPTS [FOR ALL noise, linfailin: ArrayB, ind1, ind2: Integer]
           addGood(new, noise, linfailin, ind1, ind2)
           countGood(new, noise, linfailin, ind1, ind2)
           meanSlope(new, noise, linfailin)
           sigmaS(new, noise, linfailin)

LinOut: TRAIT
ASSUMES Scale
INTRODUCES linOut: ArrayR, ArrayR, ArrayR, ArrayR, ArrayR, ArrayR,
                   ArrayB, ArrayR -> ArrayR
CONSTRAINS linOut SO THAT FOR ALL [linoffset, gm, slope0, slope1, slope2,
           temp, rawlin: ArrayR, linfailin: ArrayB, i: Integer]
      read(linOut(linoffset, gm, slope0, slope1, slope2, temp, linfailin,
           rawlin), i) = IF ~read(linfailin, i) THEN
                         read(linoffset, i) +
                         read(slope(slope0, slope1, slope2, temp), i) *
                               toVoltage(read(rawlin, i))
                                     ELSE BADDAT
```

```
% $Header: faultdet,v 1.11 86/10/06 22:33:18 tai Exp $
```

% FAULT-DETECTION MODULE

```
MapEdgetoFace: TRAIT
ASSUMES Pair WITH [Pairsort FOR C]
INTRODUCES mapE2F: Card -> Pairsort
CONSTRAINS mapE2F SO THAT
       mapE2F(1) = <1, 2>
       mapE2F(2) = <1, 3>
       mapE2F(3) = <1, 4>
       mapE2F(4) = <2, 3>
       mapE2F(5) = <2, 4>
       mapE2F(6) = <3, 4>

MapFacetoEdge: TRAIT
ASSUMES Triple WITH [Tripsort FOR C] %p.70
INTRODUCES mapF2E: Card -> Trisort
CONSTRAINS mapF2E SO THAT
       mapF2E(1) = <1, 2, 3>
       mapF2E(2) = <1, 4, 5>
       mapF2E(3) = <2, 4, 6>
       mapF2E(4) = <3, 5, 6>

MapFacetoSensor: TRAIT
ASSUMES Pair WITH [Pairsensor FOR C]
INTRODUCES mapF2S: Card -> Pairsensor
CONSTRAINS mapF2S SO THAT
       mapF2S(1) = <1, 2>
       mapF2S(2) = <3, 4>
       mapF2S(3) = <5, 6>
       mapF2S(4) = <7, 8>

Vector0: TRAIT
ASSUMES Summation WITH [vector FOR ArrayR]
INTRODUCES vector0: -> vector
ASSERTS size(vector0) = (upper(vector0) - lower(vector0)) + 1
       % no hole
       sum(vector0, upper(vector0), lower(vector0)) = 0
       % every slot is zero value

Vector0: TRAIT      % An alternative which is less formal but more readable
ASSUMES ArraySpec1 WITH [vector FOR Array, Card FOR Val]
INTRODUCES vector0: -> vector
CONSTRAINS vector0 SO THAT FOR ALL [ind: Card]
      read(vector0, ind) = 0

InnerProd: TRAIT
ASSUMES Summation, RealSpec
INTRODUCES arrayMul: ArrayR, ArrayR -> ArrayR
         innerProd: ArrayR, ArrayR -> Real
CONSTRAINS arrayMul, innerProd SO THAT FOR ALL [t1, t2: ArrayR, ind: Integer]
      read(arrayMul(t1, t2), ind) =
```

```
            read(t1, ind) * read(t2, ind)
      innerProd(t1, t2) = sum(arrayMul(t1, t2), lower(t1), upper(t1))

VotedLinout: TRAIT
ASSUMES ArraySpecR, MapFacetoSensor
INTRODUCES votedLout: ArrayR, ArrayR -> Matrix
CONSTRAINS votedLout SO THAT FOR ALL [linout, normface: ArrayR, fid: Card]
      read(row(votedLout(linout, normface), fid), 1) =
                                 read(linout, mapF2S(fid).first)
      read(row(votedLout(linout, normface), fid), 2) =
                                 read(linout, mapF2S(fid).second)
      read(row(votedLout(linout, normface), fid), 3) =
                                 read(normface, fid)

ProjDiff: TRAIT
ASSUMES MapEdgetoFace, InnerProd, VotedLinout
INTRODUCES projDiff: ArrayR, ArrayR, Matrix, Card -> Real
CONSTRAINS projDiff SO THAT FOR ALL [linout, normface: ArrayR,
                                 edgevec: Matrix, eid: Card]
      projDiff(linout, normface, edgevec, eid)
            = innerProd(row(M2I(votedLout(linout, normface)),
                  mapE2F(eid).first), row(edgevec, eid)) -
                        innerProd(row(M2I(votedLout(linout, normface)),
                                 mapE2F(eid).second), row(edgevec, eid))
% M2I is an operator which takes voted linout as argument and outputs a
% in I-Frame.  edgevec is an operator which returns a "constant maxtrix"

EdgeVectorTest: TRAIT
ASSUMES ProjDiff, LinNoise, Psqrt, MapFacetoEdge
INTRODUCES badEdge: ArrayR, ArrayB, ArrayB, ArrayR, Matrix, Card -> Bool
         EVTest: ArrayR, ArrayB, ArrayB, ArrayR, Matrix, Card -> Bool
CONSTRAINS badEdge, EVTest SO THAT FOR ALL [noise, linfailin: ArrayB,
                  linout, normface: ArrayR, edgevec: Matrix,
                                       sigmat: Real, eid: Card]
      badEdge(linout, noise, linfailin, normface, edgevec, eid)
            = read(noise, mapF2S(mapE2F(eid).first).first) |
              read(noise, mapF2S(mapE2F(eid).first).second) |
              read(noise, mapF2S(mapE2F(eid).second).first) |
              read(noise, mapF2S(mapE2F(eid).second).second) |
              read(linfailin, mapF2S(mapE2F(eid).first).first) |
              read(linfailin, mapF2S(mapE2F(eid).first).second) |
              read(linfailin, mapF2S(mapE2F(eid).second).first) |
              read(linfailin, mapF2S(mapE2F(eid).second).second) |
              abs(projDiff(linout, normface, edgevec, eid)) >
                   sqrt(2) * sigmat)
      EVTest (linout, noise, linfailin, normface, edgevec, fid)
            = badEdge(linout, noise, linfailin, normface, edgevec,
                   mapF2E(fid).first) &
              badEdge(linout, noise, linfailin, normface, edgevec,
                   mapF2E(fid).second) &
              badEdge(linout, noise, linfailin, normface, edgevec,
                   mapF2E(fid).third)

SysStatus: TRAIT
ASSUMES EdgeVectorTest, MatrixSpec WITH [Real FOR Val]
INTRODUCES countBadface: Card, Card, ArrayB, ArrayB, ArrayR, Matrix -> Card
         sysStatus: ArrayB, ArrayB, ArrayR, Matrix -> Bool
```

```
CONSTRAINS countBadface, systatus SO THAT FOR ALL [fid, fid1: Card,
                    noise, linfailin, normface: ArrayR, edgevec: Matrix]
      countBadface(fid, fid1, linout, noise, linfailin, normface, edgevec) =
            (IF EVTest(fid, linout, noise, linfailin, normface, edgevec)
                  THEN 1 ELSE 0) + IF (fid = fid1) THEN 0
                  ELSE countBadface(fid + 1, fid1,
                        linout, noise, linfailin, normface, edgevec)
      sysStatus(linout, noise, linfailin, normface, edgevec) =
            (countBadface(lower(linout), upper(linout), linout, noise,
                  linfailin, normface, edgevec) =< 2)

Even: TRAIT
ASSUMES DivMod
INTRODUCES even: Card -> Bool
CONSTRAINS even SO THAT FOR ALL [n: Card]
      even(n) = ((n mod 2) = 0)

OneorTwo: TRAIT
ASSUMES Even
INTRODUCES 1or2: Card -> Card
CONSTRAINS 1or2 SO THAT FOR ALL [sid: Card]
      1or2(sid) = IF even(sid) THEN 2 ELSE 1

SetUp: TRAIT
ASSUMES ArraySpecR, ArraySpecB, Mapsensortoface, OneorTwo,
                    Matrix WITH [Real FOR Val], EdgeVectorTest
INTRODUCES setVet: ArrayR, ArrayB, ArrayB, ArrayR, Matrix -> ArrayR
          setMat: Matrix, Matrix, Matrix, Matrix, ArrayR,
                    ArrayB, ArrayB, ArrayR, Matrix -> Matrix
CONSTRAINS setVet, setMat SO THAT FOR ALL [noise, linfailin: ArrayB, linout,
                          normface: ArrayR, edgevec, tisA, tisB,
                                tisC, tisD: Matrix, sid: Card]
      read(setVet(linout, linfailin, noise, normface, edgevec), sid) =
            IF EVTest(senface(sid), linout, linfailin, noise,
                          normface, edgevec)
                    THEN 0 ELSE read(M2S(linout), sid)
      row(setMat(tisA, tisB, tisC, tisD, linout,
                    noise, linfailin, normface, edgevec), sid) =
            IF EVTest(linout, noise,
                    linfailin, normface, edgevec, senface(sid))
                          THEN vector0
                          ELSE IF senface(sid) = 1 THEN row(tisA, 1or2(sid))
                          ELSE IF senface(sid) = 2 THEN row(tisB, 1or2(sid))
                          ELSE IF senface(sid) = 3 THEN row(tisC, 1or2(sid))
                          ELSE IF senface(sid) = 4 THEN row(tisD, 1or2(sid))

ArraySquare: TRAIT
ASSUMES InnerProd, Matrix WITH [Real FOR Val]
INTRODUCES arraySq: ArrayR, Matrix, ArrayR -> ArrayR
CONSTRAINS arraySq so that for all [f, y: ArrayR, c: Matrix, sid: Card]
      read(arraySq(f, c, y), sid) =
            square(innerProd(f, row(c, sid)) - read(y, sid))

Extremes: TRAIT
ASSUMES RealSpec
INTRODUCES
      posiInfinity: -> Real
```

```
      negaInfinity: -> Real
CONSTRAINS posiInfinity, negaInfinity SO THAT FOR ALL [r: Real]
      posiInfinity >= r
      negaInfinity <= r

Random: TRAIT
ASSUMES ArraySpecR, Extremes
INTRODUCES everyF: -> ArrayR
CONSTRAINS everyF SO THAT FOR ALL [ind: Card]
      read(everyF, ind) <= posiInfinity
      read(everyF, ind) >= negaInfinity

SpecificForce: TRAIT
ASSUMES SetUp, ArraySquare, Random
INTRODUCES bestFit: ArrayR, ArrayB, ArrayB, ArrayR, Matrix -> ArrayR
CONSTRAINS bestFit SO THAT FOR ALL [linout, normface: ArrayR,
            linfailin, noise: ArrayB, tisA, tisB, tisC, tisD,
                                        edgevec: Matrix]
      sum(arraySq(bestFit(linout, linfailin, noise, normface, edgevec),
          setMat(), setVet()), lower(linout), upper(linout)) =<
            sum(arraySq(everyf: -> ArrayR, setMat(), setVet()),
                        lower(linout), upper(linout))

LinFailOut: TRAIT
ASSUMES Specificforce, Sysstatus
INTRODUCES linFailOut: Card, ArrayR, ArrayB,
                                ArrayB, ArrayR, Matrix -> ArrayR
CONSTRAINS linFailOut SO THAT FOR ALL [sid, nsigt: Card,
            linfailin, noise: ArrayB, normface, linout: ArrayR,
                                        edgevec: Matrix, sigmat: Real]
      read(linFailOut(linout, linfailin, noise, normface, edgevec,
            nsigt, sigmat), sid) = IF sysStatus() THEN
                  IF EVTest(linout, noise, linfailin, normface, edgevec,
                        senface(sid)) THEN
                  read(linfailin, sid) | read(noise, sid) |
                  (abs(read(linout, sid) -
                          read(vI2M(bestFit(), senface(sid)), lor2(sid)))
                        > sigmat) ELSE FALSE ELSE TRUE
```

```
% $Header: estimation,v 1.9 86/10/08 22:14:05 tai Exp $
```

% ESTIMATION MODULE

```
DecodeChanFace: TRAIT
ASSUMES Pair WITH [Pairsort FOR C]
INTRODUCES decCh: Card -> Pairsort
CONSTRAINS decCh SO THAT
       decCh(1) = <1, 2>
       decCh(2) = <1, 3>
       decCh(3) = <1, 4>
       decCh(4) = <2, 3>
       decCh(5) = <2, 4>
       decCh(6) = <3, 4>

MapChanFace: TRAIT
ASSUMES SysStatus
INTRODUCES chanFace: ArrayR, Card -> Card
          nonOpFace: ArrayR -> Card
CONSTRAINS nonOpFace, chanFace SO THAT FOR ALL
                          [code, cid: Card, linfailout: ArrayB]
       nonOpFace(linfailout)
           = (IF read(linfailout, 1) & read(linfailout, 2) THEN 1 ELSE 0) +
             (IF read(linfailout, 3) & read(linfailout, 4) THEN 1 ELSE 0) +
             (IF read(linfailout, 5) & read(linfailout, 6) THEN 1 ELSE 0) +
             (IF read(linfailout, 7) & read(linfailout, 8) THEN 1 ELSE 0)
       chanFace(linfailout, 1) = IF nonOpFace(linfailout) = 1 THEN
                          IF read(linfailout, 1) & read(linfailout, 2)
                                THEN 0 ELSE
                          IF read(linfailout, 3) & read(linfailout, 4)
                                THEN 2 ELSE
                          IF read(linfailout, 5) & read(linfailout, 6)
                                THEN 1 ELSE 1
                          ELSE IF nonOpFace = 2
                          THEN IF linfailout(1) & linfailout(2) &
                                linfailout(3) & linfailout(4)
                                THEN 0 ELSE
                          IF linfailout(1) & linfailout(2) &
                                linfailout(5) & linfailout(6)
                                THEN 0 ELSE
                          IF linfailout(1) & linfailout(2) &
                                linfailout(7) & linfailout(8)
                                THEN 0 ELSE
                          IF linfailout(3) & linfailout(4) &
                                linfailout(5) & linfailout(6)
                                THEN 0 ELSE
                          IF linfailout(3) & linfailout(4) &
                                linfailout(7) & linfailout(8)
                                THEN 2 ELSE 1
                          ELSE 0
       chanFace(linfailout, 2) = IF nonOpFace(linfailout) = 1 THEN
                          IF read(linfailout, 1) & read(linfailout, 2)
                                THEN 4 ELSE
                          IF read(linfailout, 3) & read(linfailout, 4)
```

```
                        THEN 0 ELSE
                    IF read(linfailout, 5) & read(linfailout, 6)
                        THEN 5 ELSE 4
                    ELSE IF nonOpFace = 2
                    IF linfailout(1) & linfailout(2) &
                        linfailout(3) & linfailout(4)
                        THEN 0 ELSE
                    IF linfailout(1) & linfailout(2) &
                        linfailout(5) & linfailout(6)
                        THEN 5 ELSE
                    IF linfailout(1) & linfailout(2) &
                        linfailout(7) & linfailout(8)
                        THEN 4 ELSE
                    IF linfailout(3) & linfailout(4) &
                        linfailout(5) & linfailout(6)
                        THEN 0 ELSE
                    IF linfailout(3) & linfailout(4) &
                        linfailout(7) & linfailout(8)
                        THEN 0 ELSE 0
                    ELSE 0
chanFace(linfailout, 3) = IF nonOpFace(linfailout) = 1 THEN
                    IF read(linfailout, 1) & read(linfailout, 2)
                        THEN 6 ELSE
                    IF read(linfailout, 3) & read(linfailout, 4)
                        THEN 6 ELSE
                    IF read(linfailout, 5) & read(linfailout, 6)
                        THEN 0 ELSE 2
                    ELSE IF nonOpFace = 2
                    IF linfailout(1) & linfailout(2) &
                        linfailout(3) & linfailout(4)
                        THEN 6 ELSE
                    IF linfailout(1) & linfailout(2) &
                        linfailout(5) & linfailout(6)
                        THEN 0 ELSE
                    IF linfailout(1) & linfailout(2) &
                        linfailout(7) & linfailout(8)
                        THEN 0 ELSE
                    IF linfailout(3) & linfailout(4) &
                        linfailout(5) & linfailout(6)
                        THEN 0 ELSE
                    IF linfailout(3) & linfailout(4) &
                        linfailout(7) & linfailout(8)
                        THEN 0 ELSE 0
                    ELSE 0
chanFace(linfailout, 4) = IF nonOpFace(linfailout) = 1 THEN
                    IF read(linfailout, 1) & read(linfailout, 2)
                        THEN 5 ELSE
                    IF read(linfailout, 3) & read(linfailout, 4)
                        THEN 3 ELSE
                    IF read(linfailout, 5) & read(linfailout, 6)
                        THEN 3 ELSE 0
                    ELSE IF nonOpFace = 2
                    IF linfailout(1) & linfailout(2) &
                        linfailout(3) & linfailout(4)
                        THEN 0 ELSE
                    IF linfailout(1) & linfailout(2) &
                        linfailout(5) & linfailout(6)
```

```
                        THEN 0 ELSE
                    IF linfailout(1) & linfailout(2) &
                        linfailout(7) & linfailout(8)
                        THEN 0 ELSE
                    IF linfailout(3) & linfailout(4) &
                        linfailout(5) & linfailout(6)
                        THEN 3 ELSE
                    IF linfailout(3) & linfailout(4) &
                        linfailout(7) & linfailout(8)
                        THEN 0 ELSE 0
                    ELSE 0

StateSpec: TRAIT    % p.4 of Modification of 5 Easy Pieces
      State ENUMERATION OF [NORMAL, ANALYTIC, UNDEFINED]

RecordSpec: TRAIT   % p.3 of Modification of 5 Easy Pieces
ASSUMES RT WITH [Real FOR Int, State FOR Bool, acceleration FOR val,
                 state FOR bool, Record FOR T]

ArraySpecRd: TRAIT
ASSUMES ArraySpec1 WITH [Record FOR Val, ArrayRd FOR Array]

SetUp1: TRAIT
ASSUMES ArraySpecR, MapSensortoFace, MapFacetoSensor, Vector0
                        MatrixSpec WITH [Real FOR Val]
INTRODUCES setVec1: ArrayR, ArrayB -> ArrayR
         setMat1: Matrix, Matrix, Matrix, Matrix, Matrix, Matrix, Matrix,
                  Matrix, ArrayB -> Matrix
CONSTRAINS setVec1, setMat1 SO THAT FOR ALL [tisA, tisB, tisC, tisD, timA,
            timB, timC, timD : Matrix, linout: ArrayR, linfailout: ArrayB,
                  sid: Card]
      read(setVec1(linout, linfailout), sid) =
            IF ~read(linfailout, mapf2s(senFace(sid)).first) &
                  ~read(linfailout, (mapf2s(senFace(sid)).second) THEN
            read(M2S(linout), sid) ELSE
            IF read(linfailout, sid) THEN 0
            ELSE read(linout, sid)
      row(setMat1(tisA, tisB, tisC, tisD, timA, timB, timC, timD,
            linfailout), sid) =
            IF ~read(linfailout, mapf2s(senFace(sid)).first) &
                  ~read(linfailout, mapf2s(senFace(sid)).second)
                  THEN IF senFace(sid) = 1 THEN row(tisA, lor2(sid))
                  ELSE IF senFace(sid) = 2 THEN row(tisB, lor2(sid))
                  ELSE IF senFace(sid) = 3 THEN row(tisC, lor2(sid))
                  ELSE row(tisD, lor2(sid))
            ELSE IF read(linfailout, sid) THEN vector0
                  ELSE IF senFace(sid) = 1 THEN row(timA, lor2(sid))
                  ELSE IF senFace(sid) = 2 THEN row(timB, lor2(sid))
                  ELSE IF senFace(sid) = 3 THEN row(timC, lor2(sid))
                  ELSE row(timD, lor2(sid))

BestEstimate: TRAIT
ASSUMES SetUp1, StateSpec, SysStatus, Random
INTRODUCES bestFit1: ArrayR, Matrix, Matrix, Matrix, Matrix, Matrix, Matrix,
                        Matrix, Matrix, ArrayB -> ArrayR
         bestEst: Bool, ArrayR, Matrix, Matrix, Matrix, Matrix, Matrix,
                  Matrix, Matrix, Matrix, ArrayB, Real -> ArrayR
```

```
          bestStatus: Bool -> State
CONSTRAINS bestFit1, bestEst, bestStatus SO THAT FOR ALL
             tisA, tisB, tisC, tisD, timA, timB, timC, timD: Matrix,
                  linfailout: ArrayB, linout: ArrayR]
       sum(arraySq(bestFit1(), setMat1(), setVec1()),
             lower(linout), upper(linout)) =<
                  sum(arraySq(everyf: -> ArrayR, setMat1(), setVec1()),
                       lower(linout), upper(linout))
       bestEst(sysstatus, linout, tisA, tisB, tisC, tisD, timA, timB,
             timC, timD, linfailout, gn) =
                  IF sysstatus THEN vecAdd(i2n(bestFit1()), gn) ELSE 0
       bestStatus(sysstatus) = IF sysstatus THEN NORMAL ELSE UNDEFINED

SetUp2: TRAIT
ASSUMES ArraySpecR, MapSensortoFace, MapFacetoSensor, DecodeChanFace,
      MapChanFace, Vector0, MatrixSpec WITH [Real FOR Val]
INTRODUCES setVec2: Card, ArrayR, ArrayB -> ArrayR
          setMat2: Card, Matrix, Matrix, Matrix, Matrix,
                   Matrix, Matrix, Matrix, Matrix, ArrayB -> Matrix
CONSTRAINS setVec2, setMat2 SO THAT FOR ALL [linout: ArrayR,
             linfailout: ArrayB, tisA, tisB, tisC, tisD,
             timA, timB, timC, timD: Matrix, sid, cid: Card]
      read(setVec2(cid, linout, linfailout), sid) =
          IF sid = mapF2S(decCh(chanFace(cid, linfailout)).first).first) |
             sid = mapF2S(decCh(chanFace(cid, linfailout)).first).second) |
             sid = mapF2S(decCh(chanFace(cid, linfailout)).second).first) |
             sid = mapF2S(decCh(chanFace(cid, linfailout)).second).second)
          THEN IF ~read(linfailout, sid) & ~read(linfailout, sid) THEN
                  read(M2S(linout), sid) ELSE
                  IF read(linfailout, sid) THEN 0
                  ELSE read(linout, sid)
            ELSE 0
      row(setMat2(cid, tisA, tisB, tisC, tisD, timA, timB, timC, timD,
             linfailout), sid) =
          IF sid = mapF2S(decCh(chanFace(cid, linfailout)).first).first) |
             sid = mapF2S(decCh(chanFace(cid, linfailout)).first).second) |
             sid = mapF2S(decCh(chanFace(cid, linfailout)).second).first) |
             sid = mapF2S(decCh(chanFace(cid, linfailout)).second).second)
          THEN IF ~read(linfailout, sid) & ~read(linfailout, sid)
                   THEN IF senFace(sid) = 1 THEN row(tisA, lor2(sid))
                   ELSE IF senFace(sid) = 2 THEN row(tisB, lor2(sid))
                   ELSE IF senFace(sid) = 3 THEN row(tisC, lor2(sid))
                   ELSE row(tisD, lor2(sid))
                  ELSE IF read(linfailout, sid) THEN vector0
                   ELSE IF senFace(sid) = 1 THEN row(timA, lor2(sid))
                   ELSE IF senFace(sid) = 2 THEN row(timB, lor2(sid))
                   ELSE IF senFace(sid) = 3 THEN row(timC, lor2(sid))
                   ELSE row(timD, lor2(sid))
            ELSE vector0

CountOperational: TRAIT
ASSUMES MapFacetoSensor, MapChanFace
INTRODUCES countOps: Card, ArrayB -> Card
CONSTRAINS countOps SO THAT FOR ALL [cid: Card, linfailout: ArrayB]
      countOps(cid, linfailout) =
          (IF read(linfailout,
            mapf2s(decCh(chanFace(cid, linfailout)).first).first)
```

```
                THEN 0 ELSE 1) +
          (IF read(linfailout,
            mapf2s(decCh(chanFace(cid, linfailout)).first).second)
                THEN 0 ELSE 1) +
          (IF read(linfailout,
            mapf2s(decCh(chanFace(cid, linfailout)).second).first)
                THEN 0 ELSE 1) +
          (IF read(linfailout,
            mapf2s(decCh(chanFace(cid, linfailout)).second).second)
                THEN 0 ELSE 1)

ChanEstimate: TRAIT
ASSUMES SetUp2, CountOperational, StateSpec, Random
INTRODUCES bestFit2: Card, ArrayR, Matrix, Matrix, Matrix, Matrix, Matrix,
            Matrix, Matrix, Matrix, ArrayB -> ArrayR
          chanEst: Card, ArrayR, Matrix, Matrix, Matrix, Matrix, Matrix,
            Matrix, Matrix, Matrix, ArrayB, Real -> ArrayR
          chanStatus: Card, ArrayB -> State
CONSTRAINS bestFit2, chanEst, chanStatus SO THAT FOR ALL
            [linout: ArrayR,
            tisA, tisB, tisC, tisD, timA, timB, timC, timD: Matrix,
                linfailout: ArrayB, cid: Card]
      sum(arraySq(bestFit2(), setMat2(),
            setVec2()), lower(linout), upper(linout)) =<
                sum(arraySq(everyf: -> ArrayR, setMat2(),
                      setVec2()), lower(linout), upper(linout))
      chanEst(cid, linout, tisA, tisB, tisC, tisD, timA, timB,
            timC, timD, linfailout, gn) =
                IF countOps(cid, linfailout) > 2
                THEN vecAdd(i2n(bestFit2()), gn) ELSE 0
      chanStatus(cid, linfailout) =
                IF countOps(cid, linfailout) = 4
                      THEN NORMAL ELSE IF countOps(cid, linfailout) = 3
                                  THEN ANALYTIC ELSE UNDEFINED
```

```
% $Header: lpint,v 1.3 86/10/08 20:16:04 tai Exp $
```

% LARCH/PASCAL INTERFACE MODULE

```
TYPE SRARRAY, FRARRAY BASE ON SORT ArrayR FROM ArraySpecR
TYPE SBARRAY BASE ON SORT ArrayB FROM ArraySpecB
TYPE CMARRAY BASE ON SORT Matrix FROM MatrixSpec WITH [Integer FOR Val]
TYPE FRARRAY, CPARRAY BASED ON SORT ArrayR FROM ArraySpecR
TYPE EAARRAY, AAARRAY BASED ON SORT Matrix FROM MatrixSpec
                                                    WITH [Real FOR Val]
TYPE STATE BASED ON SORT Record FROM RecordSpec
TYPE VSEARRAY BASED ON SORT ArrayRd FORM ArraySpecRd

PROCEDURE RSDIMU(g, SCALE0, SCALE1, SCALE2, RAWLIN, OFFRAW: SRARRAY;TEMP,
                    NORMFACE: FRARRAY; NSIGT: Integer;
                    LINFAILIN: SBARRAY; edgevec: EAARRAY;
                    tisA, tisB, tisC, tisB, timA, timB, timC, timD: AAARRAY;
                    var LINNOISE, LINFAILOUT: SBARRAY;
                    var LINOFFSET, LINOUT: SRARRAY; var SYSSTATUS: Bool;
                    var CHANFACE: CPARRAY; var BESTEST: STATE;
                    var CHANEST: VSEARRAY)
      = COMPOSITION OF LinOffset; LinNoise; VOTELINOFFSET; sigmat;
            Linout; VOTELINOUT; LinFailout; VOTELINFAILOUT; Estimate;
                    VOTEESTIMATE END
REQUIRES lower(g) = 1, upper(g) = 3
        lower(SCALE0) = 1, upper(SCALE0) = 8
        lower(SCALE1) = 1, upper(SCALE1) = 8
        lower(SCALE2) = 1, upper(SCALE2) = 8
        lower(TEMP) = 1, upper(TEMP) = 4
        lower(row(OFFRAW, r)) = 1, upper(row(OFFRAW, r)) = 50
        lower(column(OFFRAW, c)) = 1, upper(column(OFFRAW, c)) = 8
        lower(LINOFFSET) = 1, upper(LINOFFSET) = 8
        lower(LINFAILIN) = 1, upper(LINFAILIN) = 8
        lower(LINNOISE) = 1, upper(LINNOISE) = 8
        lower(NORMFACE) = 1, upper(NORMFACE) = 4
        lower(LINOUT) = 1, upper(LINOUT) = 8
        lower(row(edgevec, r)) = 1, upper(row(edgevec, r)) = 6
        lower(column(edgevec, c)) = 1, upper(column(edgevec, c)) = 3
        lower(LINFAILOUT) = 1, upper(LINFAILOUT) = 8
        lower(CHANFACE) = 1, upper(CHANFACE) = 8
        lower(row(tisA, r)) = 1, upper(row(tisA, r)) = 3
        lower(column(tisA, c)) = 1, upper(column(tisA, c)) = 3
        lower(row(tisB, r)) = 1, upper(row(tisB, r)) = 3
        lower(column(tisB, c)) = 1, upper(column(tisB, c)) = 3
        lower(row(tisC, r)) = 1, upper(row(tisC, r)) = 3
        lower(column(tisC, c)) = 1, upper(column(tisC, c)) = 3
        lower(row(tisD, r)) = 1, upper(row(tisD, r)) = 3
        lower(column(tisD, c)) = 1, upper(column(tisD, c)) = 3
        lower(row(timA, r)) = 1, upper(row(tism, r)) = 3
        lower(column(timA, c)) = 1, upper(column(timA, c)) = 3
        lower(row(timB, r)) = 1, upper(row(timB, r)) = 3
        lower(column(timB, c)) = 1, upper(column(timB, c)) = 3
        lower(row(timC, r)) = 1, upper(row(timC, r)) = 3
```

```
        lower(column(timC, c)) = 1, upper(column(timC, c)) = 3
        lower(row(timD, r)) = 1, upper(row(timD, r)) = 3
        lower(column(timD, c)) = 1, upper(column(timD, c)) = 3

ACTION LinOffset
MODIFIES AT MOST [LINOFFSET]
ENSURES LINOFFSET!post = linOffset(OFFRAW, n2m(g),
                                   SCALE0, SCALE1, SCALE2, TEMP)
       lower(LINOFFSET!post) = 1, upper(LINOFFSET!post) = 8

ACTION LinNoise
MODIFIES AT MOST [LINNOISE]
ENSURES LINNOISE!post = linNoise(OFFRAW, LINSTD)
       lower(LINNOISE!post) = 1, upper(LINNOISE!post) = 8

ACTION VOTELINOFFSET(var LINOFFSET: SRARRAY; var LINNOISE: SRARRAY); EXTERNAL

ACTION Sigmat
MODIFIES AT MOST [sigmat]
ENSURES if ~allBad(LINNOISE) & ~allBad(LINFAILIN) then
             sigmat!post = sigmaS(slope(SLOPE0, SLOPE1, SLOPE2, TEMP),
                                  LINNOISE, LINFAILIN, LINSTD) * NSIGT

ACTION LinOut
MODIFIES AT MOST [LINOUT]
ENSURES LINOUT!post = linOut(LINOFFSET, n2m(g), SLOPE0, SLOPE1, SLOPE2, TEMP,
                     LINFAILIN, RAWLIN)
       lower(LINOUT!post) = 1, upper(LINOUT!post) = 8

ACTION VOTELINOUT(var LINOUT); EXTERNAL

ACTION LinFailout
MODIFIES AT MOST [SYSSTATUS, LINFAILOUT]
ENSURES SYSSTATUS = sysstatus(LINOUT, LINNOISE, LINFAILIN, NORMFACE, edgevec)
       LINFAILOUT!post = linfailout(LINOUT, LINFAILIN, LINNOISE, NORMFACE,
                                    edgevec, NSIGT, sigmat)
       lower(LINFAILOUT!post) = 1, upper(LINFAILOUT!post) = 8

ACTION VOTELINFAILOUT(var LINFAILOUT); EXTERNAL

ACTION Estimate
REQUIRES 1 =< cid =< 4
MODIFIES AT MOST [BESTEST, CHANEST]
ENSURES CHANFACE(cid) = chanFace(LINFAILOUT, cid)
       BESTEST.status!post = bestStatus(SYSSTATUS)
       BESTEST.acceleration!post = bestEst(SYSSTATUS, LINOUT, tisA,
              tisB, tisC, tisD, timA, timB, timC, timD, LINFAILOUT, g)
       CHANEST(cid).acceleration!post = chanEst(cid, LINOUT, tisA, tisB,
                     tisC, tisD, timA, timB, timC, timD, LINFAILOUT, g)
       CHANEST(cid).status!post = chanStatus(cid, LINFAILOUT)
       lower(CHANEST!post) = 1, upper(CHANEST!post) = 8

ACTION VOTEEST(var BESTEST: STATE; var CHANFACE: CPARRAY;
                     var CHANEST: VSEARRAY); EXTERNAL
```