Appendix D

Structured Analysis and Formal Methods

D.1 Structured Analysis and Formal Methods

Structured Analysis became popular in the 1980's and is still used by many. The analysis consists of interpreting the system concept (or real world) into data and control terminology, that is into data flow diagrams. The flow of data and control from bubble to data store to bubble can be very hard to track and the number of bubbles can get to be extremely large. One approach is to first define events from the outside world that require the system to react, then assign a bubble to that event, bubbles that need to interact are then connected until the system is defined. This can be rather overwhelming and so the bubbles are usually grouped into higher level bubbles. Data Dictionaries are needed to describe the data and command flows and a process specification is needed to capture the transaction/transformation information. The problems have been: 1) choosing bubbles appropriately, 2) partitioning those bubbles in a meaningful and mutually agreed upon manner, 3) the size of the documentation needed to understand the Data Flows, 4) still strongly functional in nature and thus subject to frequent change, 5) though "data" flow is emphasized, "data" modeling is not, so there is little understanding of just what the subject matter of the system is about, and 6) not only is it hard for the customer to follow how the concept is mapped into these data flows and bubbles, it has also been very hard for the designers who must shift the DFD organization into an implementable format.

Information Modeling, using entity-relationship diagrams, is really a forerunner for OOA. The analysis first finds objects in the problem space, describes them with attributes, adds relationships, refines them into super and sub-types and then defines associative objects. Some normalization then generally occurs. Information modeling is thought to fall short of true OOA in that, according to Peter Coad & Edward Yourdon:

1) Services, or processing requirements, for each object are not addressed,
2) Inheritance is not specifically identified,
3) Poor interface structures (messaging) exists between objects, and
4) Classification and assembly of the structures are not used as the predominate method for determining the system's objects.

This handbook presents in detail the two new most promising methods of structured analysis and design: Object-Oriented and Formal Methods (FM). OOA/OOD and FM can incorporate the best from each of the above methods and can be used effectively in conjunction with each other. Lutz and Ampo described their successful experience of using OOD combined with Formal Methods as follows: " For the target applications, object-oriented modeling offered several advantages as an initial step in developing formal specifications. This reduced the effort in producing an initial formal specification. We also found that the object-oriented models did not always represent the "why," of the requirements, i.e., the underlying intent or strategy of the software. In contrast, the formal specification often clearly revealed the intent of the requirements."

D.2 Object Oriented Analysis and Design

Object Oriented Design (OOD) is gaining increasing acceptance worldwide. These fall short of full Formal Methods because they generally do not include logic engines or theorem provers. But they are more widely used than Formal Methods, and a large infrastructure of tools and expertise is readily available to support practical OOD usage.

OOA/OOD is the new paradigm and is viewed by many as the best solution to most problems. Some of the advantages of modeling the real world into objects is that 1) it is thought to follow a more natural human thinking process and 2) objects, if properly chosen, are the most stable perspective of the real world problem space and can be more resilient to change as the functions/services and data & commands/messages are isolated and hidden from the overall system. For example, while over the course of the development life-cycle the number, as well as types, of functions (e.g. turn camera 1 on, download sensor data, ignite starter, fire engine 3, etc.) May change, the basic objects (e.g. cameras, sensors, starter, engines, operator, etc.) needed to create a system usually are constant. That is, while there may now be three cameras instead of two, the new Camera-3 is just an instance of the basic object 'camera'. Or while an infrared camera may now be the type needed, there is still a 'camera' and the differences in power, warm-up time, and data storage may change, all that is kept isolated (hidden) from affecting the rest of the system.

OOA incorporates the principles of abstraction, information hiding, inheritance, and a method of organizing the problem space by using the three most "human" means of classification. These combined principles, if properly applied, establish a more modular, bounded, stable and understandable software system. These aspects of OOA should make a system created under this method more robust and less susceptible to changes, properties which help create a safer software system design.

Abstraction refers to concentrating on only certain aspects of a complex problem, system, idea or situation in order to better comprehend that portion. The perspective of the analyst focuses on similar characteristics of the system objects that are most important to them. Then, at a later time, the analyst can address other objects and their desired attributes or examine the details of an object and deal with each in more depth. Data abstraction is used by OOA to create the primary organization for thinking and specification in that the objects are first selected from a certain perspective and then each object is defined in detail. An object is defined by the attributes it has and the functions it performs on those attributes. An abstraction can be viewed, as per Shaw, as "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary".

Information hiding also helps manage complexity in that it allows encapsulation of requirements, which might be subject to change. In addition, it helps to isolate the rest of the system from some object specific design decisions. Thus, the rest of the s/w system sees only what is absolutely necessary of the inner workings of any object.

Inheritance " defines a relationship among classes [objects], wherein one class shares the structure or behavior defined in one or more classes... Inheritance thus represents a hierarchy of abstractions, in which a subclass [object] inherits from one or more superclasses [ancestor objects]. Typically, a subclass augments or redefines the existing structure and behavior of its superclasses".

Classification theory states that humans normally organize their thinking by: looking at an object and comparing its attributes to those experienced before (e.g. looking at a cat, humans tend to think of its size, color, temperament, etc. in relation to past experience with cats) distinguishing between an entire object and its component parts (e.g., a rose bush versus its roots, flowers, leaves, thorns, stems, etc.) classification of objects as distinct and separate groups (e.g. trees, grass, cows, cats, politicians).

In OOA, the first organization is to take the problem space and render it into objects and their attributes (abstraction). The second step of organization is into Assembly Structures, where an object and its parts are considered. The third form of organization of the problem space is into Classification Structures during which the problem space is examined for generalized and specialized instances of objects

(inheritance). That is, if looking at a railway system the objects could be engines (provide power to pull cars), cars (provide storage for cargo), tracks (provide pathway for trains to follow/ride on), switches (provide direction changing), stations (places to exchange cargo), etc. Then you would look at the Assembly Structure of cars and determine what was important about their pieces parts, their wheels, floor construction, coupling mechanism, siding, etc. Finally, Classification Structure of cars could be into cattle, passenger, grain, refrigerated, and volatile liquid cars.

The purpose of all this classification is to provide modularity which partitions the system into well defined boundaries that can be individually/independently understood, designed, and revised. However, despite "classification theory", choosing what objects represent a system is not always that straight forward. In addition, each analyst or designer will have their own abstraction, or view of the system which must be resolved. OO does provide a structured approach to software system design and can be very useful in helping to bring about a safer, more reliable system.

D.3 Formal Methods - Specification Development

"Formal Methods (FM) consists of a set of techniques and tools based on mathematical modeling and formal logic that are used to specify and verify requirements and designs for computer systems and software."

While Formal Methods (FM) are not widely used in US industry, FM has gained some acceptance in Europe. A considerable learning curve must be surmounted for newcomers, which can be expensive. Once this hurdle is surmounted successfully, some users find that it can reduce overall development life-cycle cost by eliminating many costly defects prior to coding.

WHY ARE FORMAL METHODS NECESSARY?

A digital system may fail as a result of either physical component failure, or design errors. The validation of an ultra-reliable system must deal with both of these potential sources of error.
Well known techniques exist for handling physical component failure; these techniques use redundancy and voting. The reliability assessment problem in the presence of physical faults is based upon Markov modeling techniques and is well understood.

The design error problem is a much greater threat. Unfortunately, no scientifically justifiable defense against this threat is currently used in practice. There are 3 basic strategies that are advocated for dealing with the design error:

1. Testing (Lots of it)

2. Design Diversity (i.e. software fault-tolerance: N-version programming, recovery blocks, etc.)

3. Fault/Failure Avoidance (i.e. formal specification/verification, automatic program synthesis, reusable modules)

The problem with life testing is that in order to measure ultrareliability one must test for exorbitant amounts of time. For example, to measure a 10^{-9} probability of failure for a 1-hour mission one must test for more than 114,000 years.

Many advocate design diversity as a means to overcome the limitations of testing. The basic idea is to use separate design/implementation teams to produce multiple versions from the same specification. Then,

non-exact threshold voters are used to mask the effect of a design error in one of the versions. The hope is that the design flaws will manifest errors independently or nearly so.

By assuming independence one can obtain ultra-reliable-level estimates of reliability even though the individual versions have failure rates on the order of 10^{-4}. Unfortunately, the independence assumption has been rejected at the 99% confidence level in several experiments for low reliability software. Furthermore, the independence assumption cannot ever be validated for high reliability software because of the exorbitant test times required. If one cannot assume independence then one must measure correlations. This is infeasible as well---it requires as much testing time as life-testing the system because the correlations must be in the ultra-reliable region in order for the system to be ultra-reliable. Therefore, it is not possible, within feasible amounts of testing time, to establish that design diversity achieves ultra-reliability.

Consequently, design diversity can create an illusion of ultra-reliability without actually providing it.

It is felt that formal methods currently offer the only intellectually defensible method for handling the design fault problem. Because the often quoted $1 - 10^{-9}$ reliability is well beyond the range of quantification, there is no choice but to develop life-critical systems in the most rigorous manner available to us, which is the use of formal methods.

WHAT ARE FORMAL METHODS?

Traditional engineering disciplines rely heavily on mathematical models and calculation to make judgments about designs. For example, aeronautical engineers make extensive use of computational fluid dynamics (CFD) to calculate and predict how particular airframe designs will behave in flight. We use the term formal methods to refer to the variety of mathematical modeling techniques that are applicable to computer system (software and hardware) design. That is, formal methods is the applied mathematics engineering and, when properly applied, can serve a role in computer system design.

Formal methods may be used to specify and model the behavior of a system and to mathematically verify that the system design and implementation satisfy system functional and safety properties. These specifications, models, and verifications may be done using a variety of techniques and with various degrees of rigor. The following is an imperfect, but useful, taxonomy of the degrees of rigor in formal methods:

Level-1: Formal specification of all or part of the system.
Level-2: Formal specification at two or more levels of abstraction and paper and pencil proofs that the detailed specification implies the more abstract specification.
Level-3: Formal proofs checked by a mechanical theorem prover.

Level 1 represents the use of mathematical logic or a specification language that has a formal semantics to specify the system. This can be done at several levels of abstraction. For example, one level might enumerate the required abstract properties of the system, while another level describes an implementation that is algorithmic in style.

Level 2 formal methods goes beyond Level 1 by developing pencil-and-paper proofs that the more concrete levels logically imply the more abstract-property oriented levels. This is usually done in the manner illustrated below.

Level 3 is the most rigorous application of formal methods. Here one uses a semi-automatic theorem prover to make sure that all of the proofs are valid. The Level 3 process of convincing a mechanical

prover is really a process of developing an argument for an ultimate skeptic who must be shown every detail.

Formal methods is not an all-or-nothing approach. The application of formal methods to only the most critical portions of a system is a pragmatic and useful strategy. Although a complete formal verification of a large complex system is impractical at this time, a great increase in confidence in the system can be obtained by the use of formal methods at key locations in the system.

D.3.1 Formal Inspections of Specifications

Formal inspections and formal analysis are different. Formal Inspections should be performed within every major step of the software development process.

Formal Inspections, while valuable within each design phase or cycle, have the most impact when applied early in the life of a project, especially the requirements specification and definition stages of a project. Studies have shown that the majority of all faults/failures, including those that impinge on safety, come from missing or misunderstood requirements. Formal Inspection greatly improves the communication within a project and enhances understanding of the system while scrubbing out many of the major errors/defects.

For the Formal Inspections of software requirements, the inspection team should include representatives from Systems Engineering, Operations, Software Design and Code, Software Product Assurance, Safety, and any other system function that software will control or monitor. It is very important that software safety be involved in the Formal Inspections.

It is also very helpful to have inspection checklists for each phase of development that reflect both generic and project specific criteria. The requirements discussed in this section and in Robyn R. Lutz's paper "Targeting Safety-Related Errors During Software Requirements Analysis" will greatly aid in establishing this checklist. Also, the checklists provided in the NASA Software Formal Inspections Guidebook are helpful.

D.3.2 Timing, Throughput And Sizing Analysis

Timing and sizing analysis for safety critical functions evaluates software requirements that relate to execution time and memory allocation. Timing and sizing analysis focuses on program constraints. Typical constraint requirements are maximum execution time and maximum memory usage. The safety organization should evaluate the adequacy and feasibility of safety critical timing and sizing requirements. These analyses also evaluate whether adequate resources have been allocated in each case, under worst case scenarios. For example, will I/O channels be overloaded by many error messages, preventing safety critical features from operating.

Quantifying timing/sizing resource requirements can be very difficult. Estimates can be based on the actual parameters of similar existing systems.
Items to consider include:

- memory usage versus availability;
- I/O channel usage (load) versus capacity and availability;
- execution times versus CPU load and availability;
- sampling rates versus rates of change of physical parameters.

In many cases it is difficult to predict the amount of computing resources required. Hence, making use of past experience is important.

D.3.3 Memory usage versus availability

Assessing memory usage can be based on previous experience of software development if there is sufficient confidence. More detailed estimates should evaluate the size of the code to be stored in the memory, and the additional space required for storing data and scratchpad space for storing interim and final results of computations. Memory estimates in early program phases can be inaccurate, and the estimates should be updated and based on prototype codes and simulations before they become realistic. Dynamic Memory Allocation can be viewed as either a practical memory run time solution or as a nightmare for assuring proper timing and usage of critical data. Any suggestion of Dynamic Memory Allocation, common in OOD, CH environments, should be examined very carefully; even in "non-critical" functional modules.

D.3.3.1 I/O channel usage (Load) versus capacity and availability

Address I/O for science data collection, housekeeping and control. Evaluate resource conflicts between science data collection and safety critical data availability. During failure events, I/O channels can be overloaded by error messages and these important messages can be lost or overwritten. (e.g. the British "Piper Alpha" offshore oil platform disaster). Possible solutions includes, additional modules designed to capture, correlate and manage lower level error messages or errors can be passed up through the calling routines until at a level which can handle the problem; thus, only passing on critical faults or combinations of faults, that may lead to a failure.

Execution times versus CPU load and availability. Investigate time variations of CPU load, determine circumstances of peak load and whether it is acceptable. Consider multi-tasking effects. Note that excessive multi-tasking can result in system instability leading to "crashes".

D.3.3.2 Sampling rates versus rates of change of physical parameters

Analysis should address the validity of the system performance models used, together with simulation and test data, if available.