SANDIA REPORT

SAND99–1242
Unlimited Release
Printed May 1999

The Use of Object-Oriented Analysis Methods in Surety Analysis

Gregory D. Wyss, Richard L. Craft, and Donald R. Funkhouser

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (703) 605-6000
Web site: http://www.ntis.gov/ordering.htm

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A07
Microfiche copy: A01

SAND99-1242
Unlimited Release
Printed May 1999

The Use of Object-Oriented Analysis Methods in Surety Analysis

Gregory D. Wyss
Risk Assessment and Systems Modeling Department

Richard L. Craft
Information Systems Surety Department

Donald R. Funkhouser
Decision Support Systems Architectures Department

Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-0747

Abstract

Object-oriented analysis methods have been used in the computer science arena for a number of years to model the behavior of computer-based systems. This report documents how such methods can be applied to surety analysis. By embodying the causality and behavior of a system in a common object-oriented analysis model, surety analysts can make the assumptions that underlie their models explicit and thus better communicate with system designers. Furthermore, given minor extensions to traditional object-oriented analysis methods, it is possible to automatically derive a wide variety of traditional risk and reliability analysis methods from a single common object model. Automatic model extraction helps ensure consistency among analyses and enables the surety analyst to examine a system from a wider variety of viewpoints in a shorter period of time. Thus it provides a deeper understanding of a system's behaviors and surety requirements. This report documents the underlying philosophy behind the common object model representation, the methods by which such common object models can be constructed, and the rules required to interrogate the common object model for derivation of traditional risk and reliability analysis models. The methodology is demonstrated in an extensive example problem.

Acknowledgments

The authors would like to thank a number of people and organizations, without whose help the completion of this project or report would have been much more difficult. First, we would like to thank Ruthe Vandewart, who was originally a member of our research team. During the 1½ years she worked with us, she contributed immensely to our understanding of the practical aspects of both object-oriented analysis techniques and the software that is required to embody them. For this we are deeply in her debt.

We would also like to thank Sharon Daniel for her contributions to this project. She served as a sounding board for ideas many times, and provided an important resource for the project team in the area of logic methods, prime implicants, and fault tree analysis. We also had several profitable interactions with persons too numerous to mention at both the U.S. National Security Agency and at the Queensland University of Technology in Queensland, Australia. We are very grateful for their time, ideas, and cooperation.

We would also like to thank our managers at Sandia National Laboratories for their support and encouragement during this project. Allen Camp, Laura Gilliom, Judy Moore, and Bruce Malm did much to help us struggle through administrative issues as well as push us away from highly theoretical constructs and toward practical solutions to real issues.

We are also indebted to Sharon Daniel, Julie Gregory, and Ruth Haas for their review and comments, which have greatly improved the quality of this final report.

Finally, we would like to thank the Laboratory-Directed Research and Development Program at Sandia National Laboratories for the funding that provided the opportunity to do the free and creative thinking that led to the discoveries documented in this report. This work would not have been possible outside of that program, and the authors are deeply grateful for this support.

Contents

This page intentionally left blank

List of Figures

List of Tables

This page intentionally left blank

Nomenclature

API	Application programming interface
CASE	Computer-aided software engineering
CORBA	Common object request broker architecture
DFD	Data flow diagrams
ETA	Event tree analysis
FMEA	Failure modes and effects analysis
FMECA	Failure modes, effects, and criticality analysis
FTA	Fault tree analysis
GUI	Graphical user interface
HAZOP	Hazards and operability analysis
LDRD	The Laboratory-Directed Research and Development program at Sandia National Laboratories
OLE	Microsoft's object linking and embedding automation environment
OO	Object-oriented
OOA	Object-oriented analysis
OODB	Object-oriented database
OPRRA	Object-oriented process for risk and reliability analysis
RBD	Reliability block diagram
REI	Rose extensibility interface
UML	Unified modeling language

The Use of Object-Oriented Analysis Methods in Surety Analysis

1 Introduction

During recent years there has been a dramatic increase in the complexity of technology that is being used in everyday life. As complexity has increased, and as this complex technology has been applied to systems that can cause unacceptable consequences when they fail, there has been a corresponding increase in the number and severity of accidents involving such systems. These consequences may involve financial losses, compromise of information and/or security, environmental damage, injury, or the endangerment of human life. For these reasons, the importance of understanding the risks of system failure and ensuring correct system operation is increasing as well. Put most simply, the overall objective is to produce systems that do what they are supposed to do and not what they are not supposed to do; that is, build the right thing, build well, and protect it appropriately. At Sandia National Laboratories, the term "surety" was developed in the nuclear weapons program to convey this idea. Evaluating the surety of a system requires a dynamic, whole system, a whole life cycle perspective. While one might define the surety of the system in many different ways, surety often involves balancing five distinct objectives:

- Utility – correctness, or fitness for a purpose
- Integrity – completeness, validity, authenticity
- Availability – reliable, accessible and usable when and where needed
- Access control –control of the persons, places, and times when a system may be used; confidentiality, privacy
- Safety – freedom from harm to persons, property, or the environment

One key element in establishing the surety of a system is to identify and mitigate the risks of system failure. Such a goal requires a balance of potentially conflicting surety objectives. For example, one may require that an emergency system be both available for use at all times and yet accessible only to persons who are authorized to use it. A restrictive access control for the system may render it unavailable for use during a critical

time period. Risk is never zero for any system. Residual risk must be understood before its importance can be judged and a decision can be made regarding what, if anything, to do about it. An understanding of system risk will also guide assessments of how available technologies may provide increased system surety.

Historically, designers and analysts have gone about ensuring the surety of a system by examining it from many disparate perspectives using a wide variety of tools. These tools include rigorous design principles, system simulations, probabilistic risk analysis, destructive and nondestructive testing, and detailed quality assurance requirements. This "shotgun" approach has yielded very good results in certain very restrictive high-consequence environments such as aircraft, nuclear power plants, and nuclear weapons. However, this approach is very time-consuming and expensive.

Clearly, if it is to become practical for analysts to use surety analysis techniques to evaluate a wider array of systems, the approaches must be made more efficient and cost-effective. One way to accomplish this would be to build a single surety analysis model from which many analyses could be derived automatically. We have come to believe that most of the traditional risk and reliability analysis methodologies (which form the backbone of typical surety analyses) contain common, fundamental concepts which, if appropriately captured in a single entity, could operate interchangeably from a common object model. Furthermore, once this object model was constructed, one could extract from it in an automated fashion many of the traditional risk, security, and surety analysis models, including fault trees, event trees, failure modes and effects analyses, as well as discrete simulations and vital area analyses. The object model would instantiate in a formal way the analyst's understanding of the system because through the use of object-oriented analysis techniques, the analyst would encapsulate into the model the behavior and causality that make up the system.

The purpose of this report is to introduce the reader to the ways a common object model can be used to perform a surety analysis of a "real" system. Such an analysis will make use of techniques that are derived from traditional risk and reliability analysis methods. Note that each such traditional method simply encapsulates certain aspects of the behavior and causality embodied in the system using its own particular syntax and logic rules. Therefore, if the system's behavior and causality were to be embodied in a properly constructed common object model, one could, in theory, extract any type of risk or security model from the object model. This could be done *automatically* by interrogating the object model and translating the appropriate aspects of its contents into the syntax and logic rules required by the particular surety analysis method. In other words, once system behavior and causality have been embodied in the object model, the traditional risk and reliability analysis models can be obtained essentially *for free* because they are simply subsets of the object model that have been translated into another syntax.

In the fall of 1996, Sandia began conducting a multidisciplinary internal research project to create an extensible framework capable of supporting a broad range of surety assessment techniques. The project team was composed of members with backgrounds in computer security information surety, probabilistic risk and reliability assessment, and object-oriented analysis methods. This report documents the results of that study.

1.1 Historical Context

The concepts of system surety and surety analysis have only been developed during the past few decades. Prior to the 1930s, there existed very few technologies that had the potential to accidentally cause death or injury to a very large number of persons in a single event. Since that time, however, we have seen the advent of nuclear power and nuclear weapons, as well as large aircraft, complex chemical processing facilities, and other very high-consequence systems. The surety of early high-consequence systems, such as dams, was ensured by the overengineering. That is, engineering designs were endowed with a very high safety factor to ensure that catastrophic failures did not occur. A similar design philosophy was initially employed in the design of nuclear power reactors. However, systems such as large aircraft could not be designed with such high margins of safety because of significant weight restrictions (high safety margins often add weight, which severely limits the ability of the system to fly). In addition, while early systems often relied on passive safety measures, these new technologies were often forced to rely on active systems to provide safety. Obviously, active safety systems must be extremely reliable if they are to perform their intended purpose in emergency situations. Previous design and analysis methods could not provide adequate assurance that these goals were being met. Thus system designers and analysts were forced to come up with new techniques to ensure the safety of these systems in the absence of high safety margins.

In response to these requirements, several new analysis techniques were developed. Some of these methods relied on "brainstorming" to arrive at a representative list of scenarios for which the safety of the system must be examined. Other, later, techniques sought to be more systematic and exhaustive in the development of such scenarios. Fault tree analysis was developed at Bell Laboratories in the 1950s. Failure modes and effects analysis and hazards and operability analysis (HAZOP) were also developed to provide a more rigorous framework in the search for potential accident scenarios. Other methods, such as event tree and decision tree analysis, were devised to help the analyst better understand how a system might respond to a given accident condition or operator decision. The use of these methods did not become widespread until the mid-1970s, when the U. S. Nuclear Regulatory Commission sponsored the Reactor Safety Study, which used fault tree and event tree analysis to consider the safety (surety) of U. S. nuclear power plants. Throughout the 1970s and 1980s, the Nuclear Regulatory Commission was a driving force behind the development and refinement of new risk and reliability analysis methods, including the use of uncertainty analysis for surety processes. During the 1980s, analysts began to apply these same techniques in other industries, with some success. Also during that time, regulators began to require system designers and operators to perform these types of analyses for their high-consequence systems. In some cases, such analyses became a condition for licensing or certification. Similar types of requirements have been developed in the area of computer security, but historically, the techniques that have been required for these analyses have relied more on "best practices" and checklists than on the rigorous methods derived for other industries.

Also during the 1970s and 1980s, system design methodologies were becoming more rigorous and automated. The nuclear power industry introduced the concept of "defense

in depth," under which the only way that the most severe consequences could occur would require the failure of several sequential layers of defense, each of which was designed with increasing levels of conservatism. Computer-aided software engineering (CASE) tools and computerized design systems became common – especially for large companies designing complex, high-consequence systems such as aircraft. Computerized modeling and simulation began to augment and even replace system testing programs because many of the tests that would be required of new, high-consequence systems were either too expensive to perform, or involved unacceptable risks to humans or the environment. Meanwhile, the computer science field attempted several new methods to increase the reliability of software embedded in high-consequence systems. Analysts began to realize, however, that a large fraction of software errors came about, not as a result of poor coding practice, but as a result of incomplete and/or erroneous software specifications. Many of these issues are still the subject of intense research.

As we look at the evolution of design and analysis methods over the past few decades, we see that both are becoming more detailed and more rigorous. We see a relatively complete description of the system being specified in the CASE tools and computerized design systems. A complementary and increasingly detailed description of the system is also designed into the system simulation software. And the surety analyst incorporates still another detailed description of the system into his or her models so that he or she can consider whether the designed system meets all its surety objectives. Historically, each of these descriptions has been embodied in a different computer system, a different logical structure, and a different language. Communication among design, simulation, and analysis groups has been handled by interviews and the sharing of paper documents, which may be out of date before the information is actually used. This leads to inaccurate simulations, faulty surety analyses, and inadequate feedback to the design team.

Clearly it would be advantageous, both in terms of time and accuracy, to have a single up-to-date source of information regarding system design and requirements that is complete enough to be useful to design, simulation, and analysis personnel. One objective of this study was to determine the feasibility of developing such a repository using object-oriented analysis techniques.

1.2 The Need for a New Approach

The key contribution of this project has been to demonstrate that many design, simulation, and surety analysis models can be derived from a unified model if that model is carefully designed to contain the appropriate information. We have found that while each domain or technology tends to support its own specific set of design, simulation, and analysis tools, most of these should be derivable from a single knowledge repository (a unified model of the system). This unified model then would become a sort of "causal graph" that makes explicit the causes and effects found within the system. Most important, it would provide a link between those causes and effects and allow them to be traced in many different ways, depending on the particular interests being served. This causal graph could then be used as a sort of simulation tool that models the response of

the system to given stimuli. It could also be used as a deductive logic tool with which an analyst could search for the potential causes of a given system response. This would indeed be a very powerful modeling tool.

What would such a unified model look like? In short, it must make explicit everything the analyst knows about the system. It must embody the purpose of the system – it must be possible to use the model to determine whether the system is behaving in a proper or improper manner, and if improper, it must provide a measure of the consequence of that operation. It must also embody the physical composition of the system – the components that make up the system, as well as the known failure modes and vulnerabilities of individual components. It must detail how the system interacts with its environment – both the normal, expected interactions, and the abnormal, potentially challenging interactions. These interactions may be either natural or human-induced. If the system changes over time, these changes must also be incorporated into the unified model. Finally, in order to be truly complete, the model would have to examine how possible failure modes or vulnerabilities might be introduced into the system at times other than during normal operation (design, manufacture, construction, etc.). These five viewpoints of the system are discussed in some detail later in this report. Obviously, designing such a unified model that would be useful for design, simulation, and surety analysis is a very challenging task. During this project, we have made substantial progress in this regard, specifically in the areas of functionality, physical composition, and environment. The long-term realization of this vision of a unified model would result in nothing less than a revolution in the area of engineering design and analysis.

1.3 Overview of the Report

This report has nine chapters. Chapter 2 contains an overview of the historical methodologies that have been used for surety analysis. These include probabilistic risk and reliability analysis models, as well as object-oriented analysis models such as those that are popular in the computer science community. This overview is deliberately broad because we believe that most readers of this report are unlikely to be familiar with both probabilistic risk and reliability analysis modeling methods and object-oriented analysis methods. For this reason, Chapter 2 was written to enable persons who may be familiar with either of these methodologies to become reasonably conversant in the other so that they can understand the remainder of the report.

Following this overview of historical methodologies, Chapter 3 provides background to help one understand how they might go about ensuring the surety of a system. This involves understanding a system in the context in which that system is supposed to operate. One must understand a system from five separate viewpoints, including its functionality, its physical composition, the environment in which it is supposed to operate, how the system changes over time, and the system's life cycle. The concepts embodied in these five viewpoints are discussed from an object-oriented analysis perspective.

Chapters 4 and 5 provide the nuts and bolts of how one goes about building an object-oriented risk analysis model. The theory of constructing an object model is presented in

Chapter 4. This includes the basic methods by which one constructs an object-oriented risk analysis model. Chapter 5 then expands on this theory to show how various surety models can be extracted from an object model that is constructed using the techniques described in Chapter 4. These methods are illustrated using example problems in Chapter 6, while Chapter 7 illustrates how these methods can be embodied in a software tool. Finally, Chapters 8 and 9 describe the limitations that have been identified for this method. They contain suggestions for future research as well as a summary and conclusions of this study.

2 Historical Methodologies

An important objective of this project has been to demonstrate that one can bring together the previously disparate techniques used in probabilistic risk and reliability assessment and those from the world of object-oriented software analysis and design. Thus this document is intended for readers in both audiences, and it is unlikely that the reader will be familiar with the techniques found in both of these rather large toolboxes. Therefore, in order to place all readers on an equal footing, this chapter provides basic background on the tools used in both disciplines.

2.1 Risk and Reliability Analysis

If one were to survey all of the methods that are used to perform what authors call "risk and reliability analysis," they would include simple parts testing and nondestructive testing, system design and redundancy "best practices," contingency planning, and probabilistic model-based approaches. While the former methods provide useful guidance in many situations, they will not be the focus of this discussion. In order to model the reliability of and the risks associated with a complex system whose failure may cause extreme consequences, one will likely need to employ probabilistic model-based analysis tools. This section provides a basic overview of several techniques that fall into that category.

2.1.1 Failure Modes and Effects Analysis

The failure modes and effects analysis[1] (FMEA) technique, along with its close cousins failure modes, effects and criticality analysis (FMECA) and HAZOP,[2] are generally the first systematic risk and reliability analysis techniques applied to a system. All of these techniques can be classified as "inductive risk assessment methods" because they start with the definition of potential risk scenarios, and proceed to identify any risks or consequences that might occur as a result of that scenario. The potential risk scenarios are identified through both formalized methodologies and imaginative thinking, and are based on component failures, subsystem failures, human actions, and/or natural and man-made phenomena.

The purpose of a FMEA is to examine individual components and assess the effect of their failure on the system in which they are used and on other systems and subsystems. FMEA is a qualitative method that is typically documented in a tabular format. To accomplish a FMEA, the analyst goes through the components of a system one by one, and for each component considers every known failure mode individually. The analyst writes down a description of the failure mode itself, the method by which that failure would be detected in the operating system, the effect of the failure on the system or subsystem, and the expected response of operators or automatic controls to the situation. Elucidating comments are also included in order to allow others to understand the full scope and gravity of the situation caused by the component's failure.

The FMEA table is often extended by including extra information in the analysis table. Typical extensions include a quantitative or qualitative estimate of the likelihood of each assessed component's failure, a qualitative categorization of the criticality of the effects caused by the component failure, and the possible actions to reduce the failure rate or effects. This is often called a FMECA. In such an analysis, one can rank the results in terms of either the likelihood or the criticality of the component failure scenarios. A more recent ranking method combines the likelihood and criticality descriptors to obtain a "risk descriptor" that is low for improbable, low-criticality events, and increases as either or both of the descriptors become large. These risk descriptors can then be used as a basis for determining whether remedial action should be taken to reduce the likelihood or criticality of the scenario.

A HAZOP study is related to a FMEA or FMECA in that it assesses predefined scenarios to determine their probable causes, consequences, and possible remediation actions. It also typically includes qualitative assessments of criticality, likelihood, and risk similar to those described above. However, while FMEA and FMECA studies focus on the effects of individual component failures, the HAZOP method focuses on qualitative deviations of key system operating parameters from their nominal, normal, or design values. The fundamental philosophy here is that normal operations are inherently safe, and deviations are the source of unrecognized problems. The scenarios that can lead to these deviations are arrived at through a combination of systematic consideration, deductive logic (to obtain probable causes for the parameter deviations), and imaginative thinking. The objective is to find the "weak link" in the system, and to provide a basis for developing procedural or engineering controls to reduce any risks so identified.

The most important limitations of these techniques are related to their reliance on a "bottom-up" problem-solving method. By this we mean that the sources of risk are identified at the *beginning* of the analysis, instead of being inferred by a systematic deductive "top-down" analysis such as would occur in a fault tree analysis. If the risk analyst does not think of a particular scenario, and the mechanics of the analysis or the "best practice" checklist does not drive them to identify it, then that scenario is likely to remain unanalyzed because the scenarios are the starting point of the analysis, *not* its result. In addition, the one-by-one nature of parameter variation in a HAZOP study and failure consideration in a FMEA or FMECA can neglect the effects of multiple concurrent failures or variations, which may have both significant likelihood and high criticality.

2.1.2 Event Tree Analysis

Event tree analysis[3] (ETA) is an inductive risk and reliability assessment technique that seeks to represent an undesired occurrence as a sequence of events. Event trees are similar in form to decision trees, and are used to represent the spectrum of possible outcomes given a particular initial condition. The method is inductive in that it begins with a particular set of initial conditions and uses inductive logic rather than deductive logic to infer its results. Each path through the event tree is constructed by selecting a unique outcome for each event within the event tree model. Thus the path physically

represents a unique sequence of events so that outcome O_{1P} occurs for event 1, *and* outcome O_{2P} occurs for event 2, *and* outcome O_{3P} occurs for event 3, and so forth. If the event tree model is properly constructed, the set of all paths through the model represents the complete set of possible outcomes that can occur as a result of the given initial condition (but typically only the outcomes relevant to the analyst's needs).

The events within an event tree may include the status of physical systems, operator actions, the activities of automated control systems, and random (stochastic) events both internal and external to the system. The events may represent simple yes-or-no questions ("binary events" such as, "Does the operator turn the system on?"), or they may involve multiple possible outcomes ("multibranch events" such as, "Which of the five displays does the operator check first?"). The events may or may not be independent of one another. If the events are not independent, then the dependencies between them are explicitly included within the logical structure of the tree.

The results of an event tree analysis are initially qualitative in that each path defines a scenario in terms of the outcomes for individual events. If, however, one assigns conditional probabilities to the various outcomes such that $P(O_{2P})$ is actually the conditional probability that O_{2P} occurs *given* that O_{1P} has already occurred, then one can also obtain quantitative results consisting of the scenario (path) definition and its probability of occurrence.

An event tree is by definition an acyclic graph. Since cycles are prohibited, it can be difficult to represent the behavior of systems that embody feedback loops in an event tree model.

2.1.3 Fault Tree Analysis

Fault tree analysis[4] (FTA) is a risk and reliability assessment technique that uses deductive reasoning, as expressed by graphical logic diagrams, to determine how a particular undesired event can occur. The purpose of the graphical logic diagram is to illustrate the individual steps in the deductive reasoning process so that others can understand, not only the results (*how* and *why* things fail), but also the method by which those results were obtained (why *these* elements contribute to system failure). The logic diagram is constructed using the method of *immediate cause* in which one finds the immediate, necessary, and sufficient conditions for each deductive logical step to be satisfied. This method, also known as the "rule of small steps," helps ensure the logical completeness of the fault tree model by ensuring the completeness of the logic at each small step. The premise is that by being logically complete at each small logical step, and allowing the overall logic of the fault tree model to dictate the assembly of these individual logical statements, one has some confidence in the completeness of the overall logical model.

Once the fault tree logic diagram is constructed, it is solved to find the *minimal cut sets*. Each minimal cut set represents one set of necessary and sufficient conditions for the occurrence of the undesired event that the fault tree was constructed to investigate

(system failure, for example). It is, in essence, a definition of one scenario that results in system failure. The overall group of minimal cut sets then represents the universe of possible scenarios that will lead to this undesired event. These *qualitative* results can then be used to provide *quantitative* insights because when a probability of occurrence is associated with each basic event in each cut set, we can determine an overall probability for each scenario and rank the scenarios accordingly. Furthermore, one can dissect the cut set results using some simple mathematical manipulations to determine the importance of individual basic events to the overall risk performance of the system.

While FTA is a very structured and systematic way to assess a single system, it can also accurately represent the interactions among multiple systems. It is not unusual to model complex interactions among systems using event trees in which the causes for each event in the event tree are determined using FTA. Such a combination of fault trees and event trees also works well when an operator must perform a procedure that initiates or terminates the operation of more than one system. In each case, the event tree paths that lead to undesired outcomes are used to construct logic that links the various system fault trees together into a global fault tree model that embodies the dependencies between the systems.

FTA is one of the few techniques that adequately treat common mode and common cause failures. In addition, it allows the analyst to consider the effects of human operators and automatic control systems on these individual failure scenarios through the application of "recovery events" to cut sets on a case-by-case basis. The principal drawbacks to FTA are that it often does a poor job of representing time-dependent scenarios, and that it can be quite time consuming and expensive to apply. In addition, since a fault tree is by definition an acyclic graph, it can be difficult to represent the behavior of systems that embody feedback loops or other circular dependencies in a fault tree model.

Another method that is related to FTA is that of the reliability block diagram[1] (RBD). An analyst constructs an RBD by simply following a flow chart for a system from all inputs to all outputs. The objective of such a reliability analysis is to determine all of the ways that the flow of elements through a system can be disrupted. The analyst must be careful to ensure that the flow chart block diagram accurately represents all of the series and parallel paths through the system. The flow chart can then be converted into a fault tree by viewing every parallel path as a logical AND gate (all of the parallel paths must fail simultaneously) and every series path as a logical OR gate (any single failure in the series can disable that path). This process is more intuitive than FTA for persons who are not experienced in probabilistic risk and reliability techniques, and since it is logically equivalent to FTA, produces the same types of results. Its biggest drawback compared with FTA is that complex logical relationships that are easily expressed in FTA may be difficult or impossible to represent clearly using the flow chart graphics implied by the RBD method.

2.1.4 Influence Diagrams

An influence diagram[5] is an acyclic probabilistic network that consists of nodes and arcs. The nodes can represent system states, decisions, or chance or deterministic occurrences, while the arcs represent the conditional dependencies among these occurrences. The nodes ultimately influence a "value node" that quantifies the consequences for each possible combination of occurrences and system states. Conditional probabilities can be applied within various nodes to represent the probability that a particular event happens *given* particular conditions in the other nodes to which it is connected (i.e., states, decisions, or events that *influence* this node). Thus an influence diagram consists of four distinct parts: the nodes, the influences upon the nodes (the dependencies among the nodes, as represented by the arcs), the conditional dependencies within each node upon other nodes in the model, and the conditional probabilities themselves.

The influence diagram method is conceptually similar to the event tree, decision tree, and fault tree methods described earlier. It can be applied as both an inductive and a deductive modeling tool in that one can begin either with the value node (the objective, as is done with fault tree analysis) or with a suitable initial condition (as is done with event tree and decision tree analysis). One could even begin with some of each and work both inductively and deductively as necessary until the model is complete. In addition, the method is not limited to simple binary events as is FTA. This flexibility makes the influence diagram an important tool to the risk analyst.

Historically, the most important disadvantage to the use of influence diagrams is that the traditional solution method does not show or even generate the detailed set of scenarios or paths possible in the model. The ability to examine these paths in detail is a primary advantage of the FTA and ETA methods. In addition, the traditional solution method makes it difficult to determine which nodes are the most important for various aspects of the results and hence to determine where one should look to improve the system. These results are very important in both risk and reliability analyses. However, it has been noted that there always exists a transformation from an influence diagram model into an event tree model. When applied, this allows an analyst to use the advantages associated with influence diagram construction to produce the more desirable results associated with ETA. While this transformation is not yet automated, it is straightforward and amenable to automation.

2.1.5 Markov Models

Markov models[1] are directed graphs that capture the concepts of system states and probabilistic transitions between states. To build a Markov model, an analyst examines every relevant configuration of a system – both functional and nonfunctional configurations – and defines them to be *states* of the system. The analyst then defines the probability of transition from each state to every other state (as a function of time and other factors) to complete the model. State transitions that are precluded for physical reasons are assigned a transition probability of zero.

Markov models provide a natural, direct representation, through the use of cycles, of systems whose components are repairable and systems where component failures have interactions. Recall that fault trees and event trees are acyclic graphs and hence do not readily accommodate these system characteristics. The two basic forms of Markov models are chains and processes. A Markov chain uses matrix multiplication in discrete time to obtain state transition probabilities. A Markov process uses a set of differential equations over continuous time. Relative to the other techniques discussed, Markov processes require a more sophisticated understanding of mathematics for their solution. In fact, most Markov models of real systems suffer from "state explosion" and hence are difficult to solve, requiring simulation. Complete path or scenario information is not a natural output of a Markov model.

2.2 Object-Oriented Analysis Methods

As will be described later, the assessment approach presented in this paper borrows system modeling concepts from "object-oriented analysis" (OOA). This is a family of system analysis methodologies used to identify system requirements and initial design concepts. *Objects* are model constructs used to represent real-world entities that can "communicate" with one another. This communication is considered to consist of *messages* exchanged between objects and can represent the transfer of information, materials, or energy. When an object receives a message, it responds by altering its internal *state* (e.g., the contents of a pot on a stove get hotter as the burner delivers heat to the pot) and/or by generating outbound messages (e.g., steam that is created as the pot's contents heat). The way in which the object responds to messages depends on its internal *processes* and on its internal state.

While a number of OOA approaches exist, the two most popular are the Shlaer-Mellor method and the Unified Modeling Language (UML).

2.2.1 The Shlaer-Mellor Method

Shlaer-Mellor is a methodology[6] (as opposed to UML, which is a visual modeling language). To model a system using the Shlaer-Mellor approach, the analyst begins by identifying the *objects* that populate the system and by documenting the relationship among these objects using an *entity-relationship diagram* (this establishes the information structure of the system). For each of the objects identified, the analyst determines whether the behavior of the object changes with time or is invariant. If it changes, then the analyst documents these changes using a *state transition diagram*. This diagram identifies the *events* that trigger changes in behavior and the behavior that the object exhibits as it enters its new *state*. This behavior is described using *action data flow diagrams*. To document interobject interactions, Shlaer-Mellor offers *object communication models* and *synchronous access models*. For models that grow quite large, the methodology supports the notions of *domains* that partition the model space and *bridges* that specify how objects in different domains interact. Shlaer and Mellor have intentionally structured their methodology's language to limit the ways in which a

given concept can be represented. Consequently, their methodology enables *translation* of the analysis model into design.

2.2.2 UML

Unified Modeling Language[7] represents the synthesis of older methodologies by Booch, Rumbaugh, and Jacobsen and is quickly becoming the de facto standard for visual modeling languages in the systems analysis community. While it supports many of the same modeling capabilities as Shlaer-Mellor, it currently lacks an "action language" that permits detailed specification of how objects transform input messages into output messages. At the same time, UML tends to be richer in other aspects of system modeling. For example, instead of simple state transition diagrams, UML permits the use of *state charts*. UML also provides multiple mechanisms for documenting object interactions. In addition to allowing partitioning of object space, UML supports mapping of objects to underlying mechanisms, such as computers or processes.

UML is still a work in progress. A methodology that exploits the language is in the works. The Object Management Group (an industry consortium dedicated to realizing the vision of plug-and-play distributed-object computing) has adopted UML as their modeling language and is currently working to extend it with an action language. Even Project Technologies Inc., which owns the Shlaer-Mellor methodology, has endorsed UML, with a few caveats, as the visual modeling language for the analysis community.

2.3 Simulation

One of the central issues in system assessment is the analyst's understanding of the system being assessed. While static models, such as those created using object-oriented analysis techniques, are useful, they often do not provide the analyst with needed insight without a significant amount of study. For this reason, some CASE tools permit the analyst to create "executable specifications" and to then run these specifications to gain insight into system dynamics. This simulation-based approach to analysis permits the analyst to "tweak" one part of the system and to see how the rest of the system responds. The underlying assumption in this approach is that humans are good at discerning patterns in large volumes of information and that by allowing the analyst to create "what if" situations with the simulation model, it becomes possible for the analyst to recognize some of the anomalous behaviors in the system.

2.4 References

1. McCormick, N.J., *Reliability and Risk Analysis: Methods and Nuclear Power Applications*, Academic Press, New York, 1981.

2. Greenberg, H.R., and Cramer, J.J., editors, *Risk Assessment and Risk Management for the Chemical Process Industry*, Van Nostrand Reinhold, New York, 1991.

3. Cramond, W.R., *et al.*, Probabilistic Risk Assessment Course Documentation, SAND85-1495, NUREG/CR-4350, 7 volumes. Prepared by Sandia National Laboratories for the U.S. Nuclear Regulatory Commission, Washington, DC, 1985.

4. Roberts, N.H., Vesely, W.E., Haasl, D.F., and Goldberg, F.F., Fault Tree Handbook, NUREG-0492, U.S. Nuclear Regulatory Commission, Washington, DC, 1981.

5. Jae, M., and Apostolakis, G.E., "The Use of Influence Diagrams for Evaluating Severe Accident Management Strategies," *Nuclear Technology* **99**:142-157, August 1992.

6. Shlaer, S., and Mellor, S.J., *Object Lifecycles: Modeling the World in States*, Prentice Hall, Englewood Cliffs, N. J., 1992.

7. Booch, G., Jacobson, I., and Rumbaugh, J., The Unified Modeling Language User Guide, Addison-Wesley, New York, 1998.

3 Understanding the System and its Context

The central idea advanced by this research project is the value of an explicit system model. This chapter explains the rationale behind model-based assessment and describes the nature of the system model needed to support this approach.

3.1 The Value of Model-based Assessment

Figure 3-1 depicts the normal process used in the development and analysis of assessment models. In this process, the analyst typically studies the system in question and formulates a mental model of how the system is structured and how it behaves. From this mental model, the analyst then constructs one or more assessment models (fault trees, event trees, etc.) that address the questions of concern to the analyst. Each of these models is then evaluated either by hand or with the aid of an automated tool to produce a set of findings about the system.

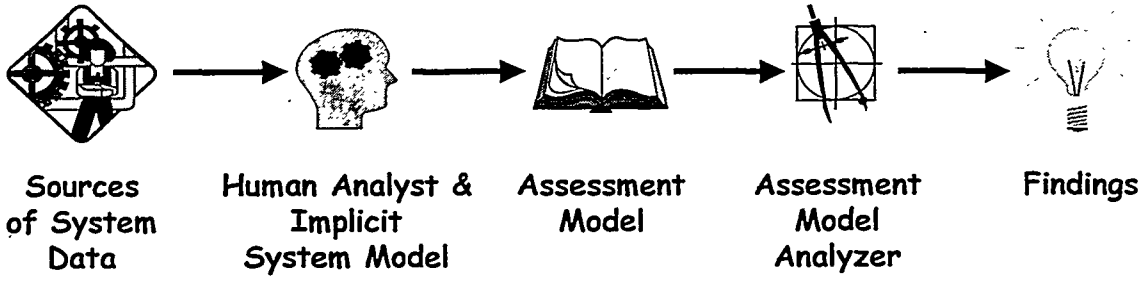


Figure 3-1. How assessment is typically done.

This approach to assessment suffers from several shortcomings,[1,2] in particular, the fact that the analyst's understanding about the system resides principally in the analyst's mind. This means that the analyst's understanding cannot be assessed independently. If elements are missing from the assessment models or are deemed to be incorrect by independent analysts, it is often unclear whether the original analyst did not know certain things about the system, knew about them but forgot them, knew about them but did not consider them significant, or simply misunderstood or misrepresented certain facts about the system. A second shortcoming is the fact that this approach requires the analyst to hand tool one or more assessment models for each of the surety issues being considered for the system. Since the assessment of most real-world systems requires that a number of questions be addressed, the construction of the entire suite of assessment models needed by the analyst proves to be a time-consuming activity. A third problem with this approach is the issue of thoroughness. For lack of an explicit system model, the analyst tends to overlook things that should find their way into the assessment model. Finally, as

long as system assessment is based on the ability of humans to produce assessment models, then the field of surety engineering will remain more craft than science.

Figure 3-2 depicts a better approach to system assessment. Using the same information sources as the first approach, the analyst focuses most of his or her effort on developing an explicit system model. This model documents everything of relevance that the analyst understands about the system. The analyst then provides this model and a set of surety questions to be addressed to a software tool that automatically produces the assessment models for each of the questions. Analysis of these models then proceeds as usual in order to produce the findings.

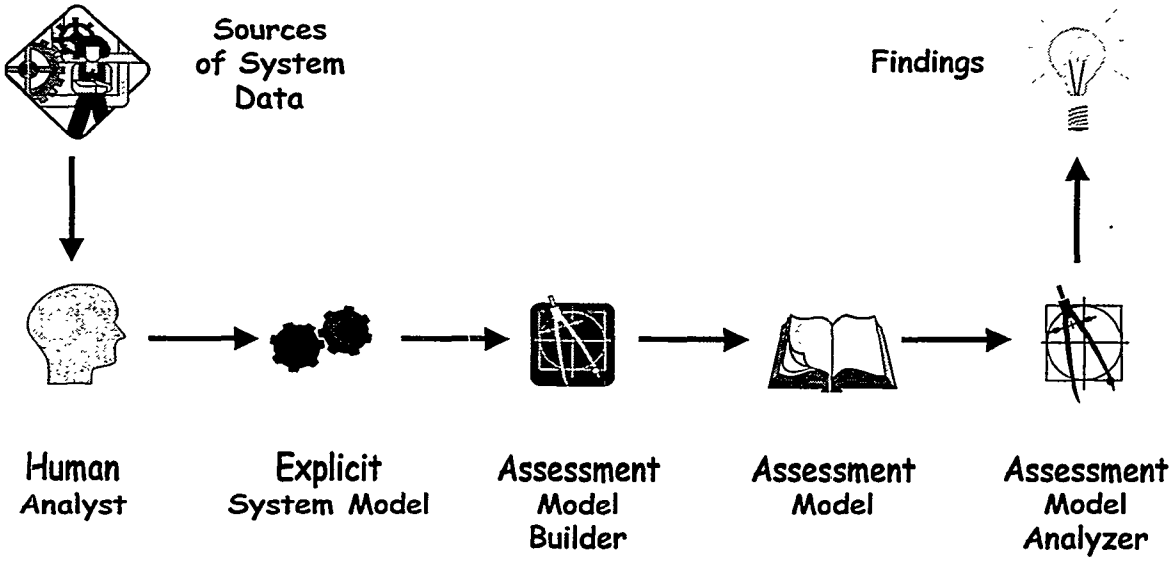


Figure 3-2. Model-based Assessment.

3.2 The Nature of the System Model

In surveying assessment techniques, it became apparent to the project team that most techniques focus on either (or both) of two questions:

- "How could the system produce result X?" or
- "What happens if event Y happens in the system?

The first question typifies the kind of reasoning used in the construction of fault trees. The second reflects the line of reasoning used in building event trees.

In answering these questions, an analyst will typically use some sort of system diagram that documents the interrelationships between components in a system (such as the piping

diagram in a chemical processing plant). To answer the "How could ..." question, the analyst starts at the point of interest indicated by the question and then traces backward against system flows to determine the possible immediate and long-term causes for the problem being analyzed. In the case of the "What happens ..." question, the analyst starts at the point associated with the event in question and traces forward, determining what immediate and ultimate effects result from Y's occurrence.

In both of these cases the primary concept being exploited is causality (i.e., the network of cause-and-effect relationships in a system), and the mechanism being used to extract this causality information is the system's structure and the behavior of its individual components. For this reason, the project team pursued a model that makes it easy for the analyst to document this sort of information about a system.

In modeling a system using the approach developed in this project, the analyst specifies the system's composition and its context. The analyst explicitly identifies those components that make up the system being assessed and those external to the system with which the system interacts. The analyst then documents the flows of energy, materials, and/or information that occur between these components. Given this specification of flows into and out of each component, the analyst then documents how each component transforms its input flows into output flows.

3.2.1 Model the "Normal System"

Figure 3-3 presents a notional view of the model used to describe system components. Each component can have inputs, outputs, processes, and an internal state. Inputs and/or the internal state drive the component's processes which, in turn, produce the component's outputs and/or alter the component's internal state.

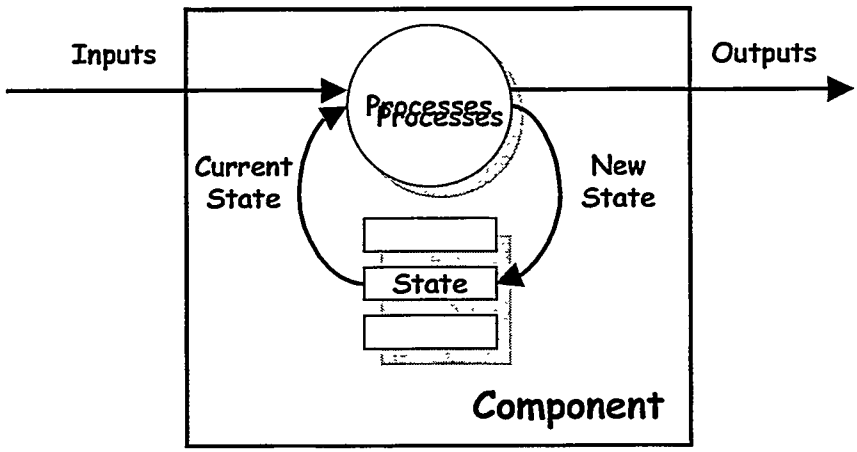


Figure 3-3. Notional View of Component Model.

3.2.2 Model Abnormal Component Behavior

Once this "normal" system model (i.e., the model that explains how the system is intended to operate) is specified, the analyst extends it with information about the ways in which individual components can fail and in which flows between components can be altered. For each input flow and each element of a component's state, the analyst considers what deviations from "normal" are possible and what effect these deviations have on the component's outputs and on its processing. Similarly, for each output flow and each element of the component's state, the analyst identifies potential deviations and traces these back to failed processes, deviations in the "current state," and deviations in the input.

3.2.3 Model Additional Component Flows

In addition to evaluating aberrations in already identified flows, the analyst considers whether each component is capable of generating or responding to other flows. For example, a computer that receives a sufficiently large mechanical shock or that exists in an excessively hot environment may cease to function. In identifying both these additional flows and deviations in normal flows, the analyst expands the causes-and-effects model for each component.

3.2.4 Model the Physical Aspects of the System

In the assessment of many systems, the issue of "place" is of critical importance. No physical system exists in isolation; rather, its components occupy particular locations in space. Computers can exist in racks and these racks reside in rooms in buildings. The pipes in a chemical plant run through specific locations at the plant site. The plant site itself can be important (e.g., whether it sits on a fault or in a location with a history of flooding). For this reason, the analyst may expand the system model to include buildings, sites, etc. that describe the system's physical environment. In doing so, the analyst may also describe the dimension, location, and orientation of both the components that populate the system and those that constitute its context, as well as the physical composition of these entities.

3.2.5 Identify Other Intercomponent Flows

At this point in the development of the system model, each component in the model presents a number of capabilities to its outside world. Said differently, the component says, "Here are all of the flows that I will respond to and here are all of the flows that I can generate." Given this, the analyst's next job is to identify other possible flows in this system. To do this, the analyst considers whether any of the components in the model can exchange flows in unanticipated ways (e.g., a leaking water pipe might be positioned in such a way as to send water into a computer). The analyst also determines whether other external entities not currently a part of the system model could interact with components in the system. For example, a custodian with access to a computer room might be able to hack a computer from its system console. If a building that houses the

system being assessed sits at the end of a runway, an airplane crash might pose a threat to the system's ability to function.

For some external entities in the system, the analyst needs to identify what other relationships must exist if the entities are to deliver or respond to certain flows. For example, the analyst may determine that a truck transporting nuclear materials between two sites is vulnerable to attack by a team of terrorists equipped with certain tools and specific pieces of knowledge (e.g., the truck's exact itinerary and schedule). In this case, the potential sources of these requisite elements become new entities in the system model and the elements become flows (e.g., the computer used to plan and track shipments would become a new entity in the system model, and the itinerary and schedule information would become flows between the computer and the terrorist team).

3.2.6 Address Temporal Aspects of the System Model

By this point in the process, the analyst has built a well fleshed-out model of the system being assessed. Even so, the model can be augmented in at least two other ways. First, as described here, many aspects of the model, such as a component's physical location in its environment or the flows among certain components, are treated as though they were static when they could actually be dynamic. For example, whether or not two mobile patrols using line-of-sight communications can talk with one another depends on the nature of the terrain that they are traversing and where each exists in that terrain. While this is an issue in real-world assessments, the project ended before significant results could be obtained for this important aspect of system modeling.

3.2.7 Model the Life Cycles of System Components

In some systems, understanding a component's life cycle (i.e., where it has been from the time it is built until the time it reaches the junk pile) is an important issue. The reason for this is that what happens to a component at one point of its life cycle can affect its behavior at a subsequent point. For instance, a computer's built-in programming can be altered during shipment from the factory to the operational site so that once the computer is in operation, it is more easily subverted by a hacker. Similarly, whether or not a hamburger is safe to eat depends on how its ingredients are handled at the various food processing plants used to create the ground beef, the buns, and so on. To model this, the analyst starts with the system under consideration and identifies the life cycle stages associated with each of the system's components. For each such stage, a system model can be constructed in a manner similar to the total system model described previously in order to assess whether the given component can be subverted prior to its inclusion in the system. Similarly, subsequent life cycle stages can be modeled as needed to address other issues, such as confidentiality (e.g., is secret information leaving the system by means of a retired computer?) or environmental safety (e.g., has a storage tank been decontaminated before going to the scrap metal dealer?).

3.3 Deriving Assessment Models

Once the system model has been constructed, assessment models can be automatically derived by software. In general, the process works as follows:

1. The analyst picks initial conditions for the system and a point within the system for analysis. For example, the analyst can select a given flow and one or more of the abnormal values that it can assume. The same could be done for the elements of a component's state or for the component's processes.

2. The analyst then tells the software which way to propagate a graph from that starting point.

3. If the analyst specifies propagation against the system flows, a deductive model such as a fault tree will be built. To do this, the software traces backward against the flow from its starting point and asks the entity from which the flow emanates how it could produce that particular starting condition. The entity then examines itself to determine which internal conditions could cause the queried condition. Any information found is used to grow the fault tree. In addition, if the entity determines that the queried condition could be created as a result of certain flows received from other entities, then the entity passes this information on to the software. In like fashion, the software asks these new entities how they could produce the conditions specified by the previous entity and each then returns explanations rooted in internal failures and in inputs received from other entities. In this way, the fault tree is built bit by bit until the only entities left to be queried are those that exist outside of the system boundary. The flows received from these outside entities (per the specification of entities inside the system) constitute primary events that populate the leaves of the fault tree.

4. If the analyst specifies propagation with the system flows, then an inductive model such as an event tree will be built. In this case, the software follows the initial flow until it reaches an entity that receives the flow. The software then asks the entity what it will do if the specified initial flow is received. The entity examines itself to determine its failure modes, along with the associated outputs. These modes and outputs are used to build that part of the event tree that corresponds to the queried entity, and the outputs are used by the software to determine which entities to pursue next. The software continues in this fashion until each output flow terminates in an entity outside of the system. Of course, variations on this theme are possible. For example, the analyst can specify a starting state for each entity in the system and ask what happens under those conditions. Also, the analyst can select two points in the system (e.g., two flows and their associated states) and ask the system to find all paths that link the initial point to the terminal point.

3.4 Other Aspects of System Modeling

So far in this chapter, the focus of the discussion has been on how to develop a model that allows causal graphs to be automatically extracted by software. In order for these

graphs to be used to produce "findings," some additional information needs to be added to the models. In particular, information regarding the relative likelihood of occurrence for different events must be factored into the assessment models. For certain types of systems, such as those based on physical devices and in which device failure represents the primary surety issue, the methods for determining and using these likelihoods are fairly well understood. For other systems, such as computer networks, specifying the likelihood of an event occurring (e.g., a given router being hacked) is much more difficult, and alternative approaches to ranking findings are needed. In either case, these sorts of issues were considered in the course of this project, but were not the central focus of this effort.

3.5 References

1. Fletcher, S.K., Jansma, R.M., Lim, J.J., Halbgewachs, R., Murphy, M.D., and Wyss, G.D., "Software System Risk Management and Assurance," in *Proceedings of the 1995 New Security Paradigms Workshop*, IEEE Computer Society, 1995.

2. Caelli, W., Longley D., and Tickle A., "A Methodology for Describing Information and Physical Security Architectures," internal report of the Information Security Research Center of the Queensland University of Technology, Queensland, Australia, 1992.

4 The Nature of the System Models

Given the model-based approach to assessment described in the Chapter 3, the next question to address is what these systems models are like and how they are constructed. This chapter describes this modeling approach.

4.1 Overview

As already noted, the system model describes various aspects of the system being assessed. These include:

- the system's functionality
- its logical partitioning into components
- interactions between components
- the physical structure (form and material composition) of these components
- the system's environment and how the system is situated in this environment
- how all of the above change with time
- the life cycles of each of the system's components

Each of these aspects of the system is modeled by creating "views" of the system. Each such view typically documents a limited but interrelated set of facts about the system. By virtue of the fact that different views will address common elements in the system, the views taken as a whole describe the system. Figure 4-1 illustrates this relationship between views and the underlying system model.

4.2 Documenting the System's Normal Behavior

The first step in modeling the system is to document its behavior. This consists of identifying its blocks of functionality, and documenting how the blocks interact and how they act within the context of these interactions.

4.2.1 The Functional Structure

To document the system's functionality, the methodology described in this document uses four types of views: a system structure diagram, a interaction diagram, a state transitioin diagram, and a data flow diagram. The *system structure diagram*, presents the "chunks" of functionality found in the system and specifies how these chunks relate to one another (Figure 4-2). In Figure 4-2, each of the blocks (M through P) represents some portion of the functionality delivered by system A (which could itself be a

component in some higher level system). When taken together, these blocks embody the entire functionality of system A. Lines between any two blocks indicate that those blocks interact with one another. As needed, each block in a diagram can be further subdivided. This process of hierarchical decomposition of the system is carried as far as is needed for the analyst to adequately document the functional structure of a system.

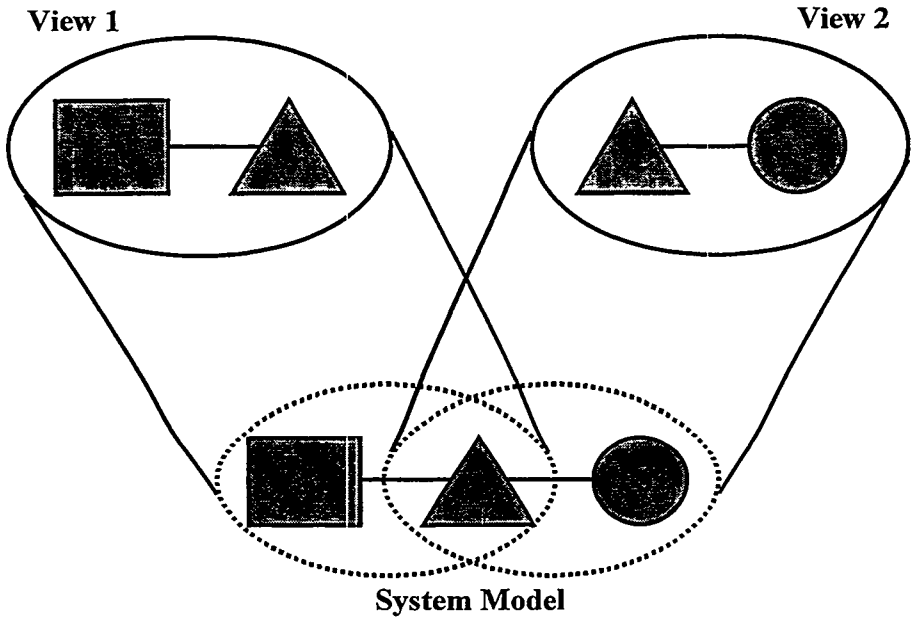


Figure 4-1. Views Into a System Model

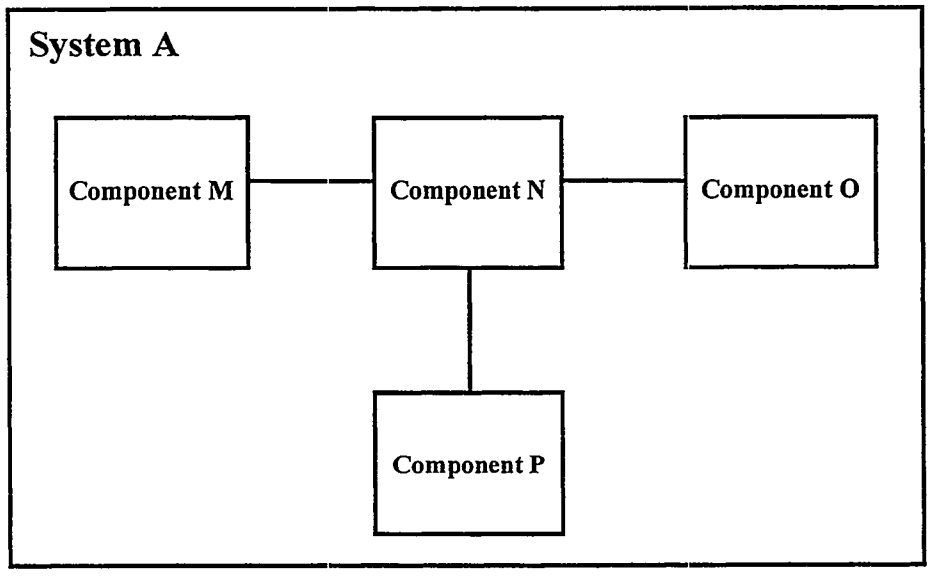


Figure 4-2. A System Structure Diagram

4.2.2 Functional Interactions

A second view used to describe a system's functionality is the *interaction diagram*, as shown in Figure 4-3. This diagram's function is to capture the dynamics of the system at "black box" levels. Using the blocks produced in the system structure diagrams, interaction diagrams show the "flows" that occur among various functional blocks in the model. These flows can represent materials (i.e., tangible things), energy, or information. Depending on the nature of the analysis to be done, the labels on the flows (e.g., A, B on the upper left flow) will identify either attributes of the flows that occur between two blocks or will represent objects that flow from block to block. For example, if component N outputs a chemical and component O consumes it, then flow A might specify attributes of the chemical that is flowing (e.g., the quantity of chemical, the chemical's temperature and composition). The result of this flow (as will be discussed below) could be to change the internal attributes (e.g., the amount of the chemical that each block stored) of components N and O. The concept here that is important from a modeling point of view is that a flow occurred that resulted in changes in the attributes associated with the components involved in the flow.

On the other hand, if the behavior of what is flowing is important, then the analyst may choose to model the flow as an object. For example, component N might represent the process that produces a sterile container of some sort. Flow B represents the containers that flow from component N to component O, which documents the process by which the containers are filled. In this case, the analyst may choose to model the containers as objects that are described by a set of attributes (e.g., capacity, contents) and behaviors (e.g., "accept contents" or "dispense contents"). Given this, a flow then becomes a framework for describing how a given object (in this case, the containers) moves from one relationship (i.e., with the process that creates it) to another (i.e., with the process in which it is filled). Finally, since the analyst builds the system model by creating a series of diagrams, a given functional block may be found in several different interaction diagrams, indicating that this block participates in multiple transactions within the system.

Figure 4-3. An Interaction Diagram

Before describing the next two diagrams, it is worth summarizing what the analyst has done by developing the first two sets of diagrams. In creating the system structure

diagrams, the analyst hierarchically decomposes the system into a set of independent functional blocks (Figure 4-4). At any given level, each block in this treelike decomposition structure is independent of every other block. In creating interaction diagrams, the analyst knits these blocks together by showing all of the flows in and out of each block (Figure 4-5). By following the flows at the lowest level of the diagram (the graph shown at the bottom of the figure), the analyst is able to identify all of the potential relationships that exist within the system. For example, flow P ***may*** be influenced by either flow C or D_2, or by both. While this information is necessary for analysis, it is not sufficient. For this reason, the analyst uses two additional sets of diagrams that document the specifics of how each functional block transforms its inputs into its outputs.

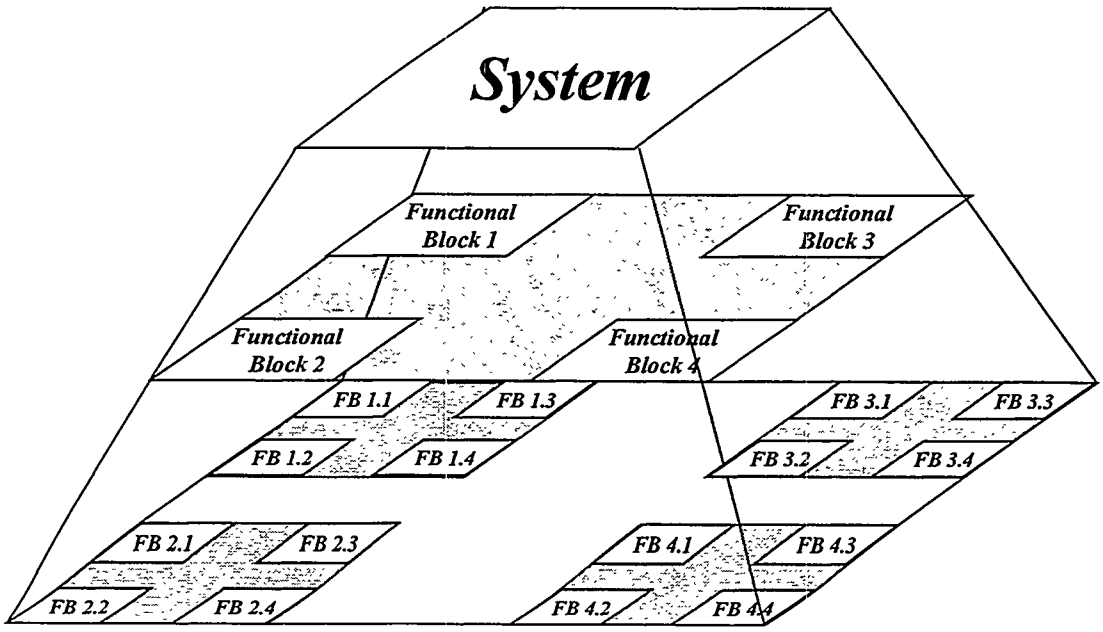


Figure 4-4. A Hierarchical Decomposition of System Function

4.2.3 Transforming Flows

If one is to truly understand the behavior of a system, one must first understand how flows are transformed within the system. This is accomplished through the use of several types of diagrams, the first of which is the *state transition diagram.* Its purpose is to document the operating states in which a given functional block can exist and to identify the events that trigger changes in the block's gross behavior (Figure 4-6). In this type of diagram, the "states" (the rounded rectangles) represent points in the life cycle of the functional block when the behavior consists of by the block consists of a unique set of responses to stimuli. The "events" (the words in italicized letters) indicate what causes

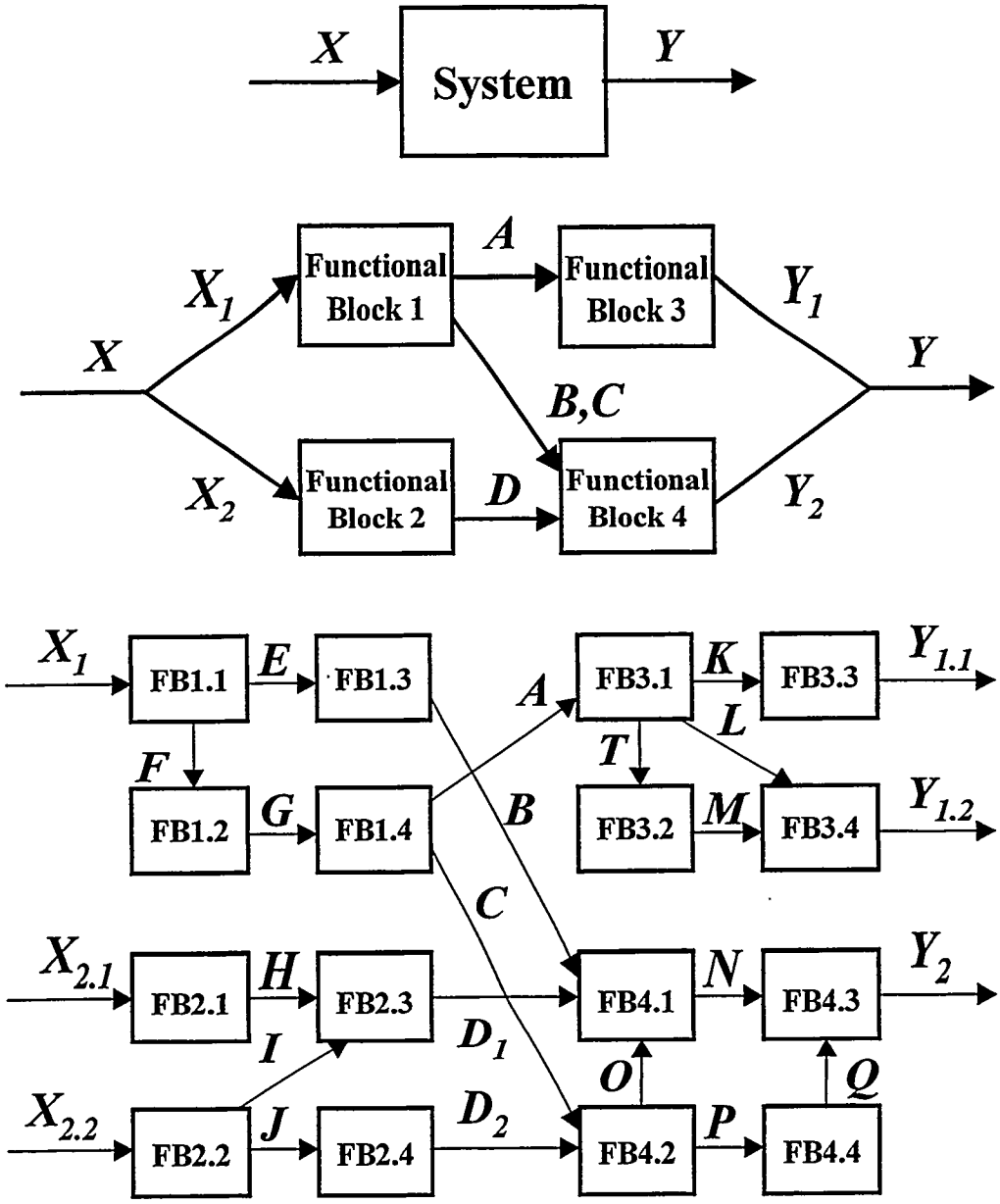


Figure 4-5. The Composite Interaction Diagrams

the functional block to transition from state to state. These may be externally generated (e.g., the arrival of a flow at a block) or internally generated (e.g., the block reaching a given internal state or a given point in time being reached). The "transitions" specify to which state a functional block moves if it receives a given event while in a given state. It should be noted that any given functional block could have multiple state-transition diagrams associated with it. The purpose of this modeling construct is to permit the analyst to capture the fact that any functional block may represent real-world processes in which multiple things can happen at once. For example, in modeling the behavior of an electrical device, one state transition diagram might be used to model the device's

purpose, while a second might be used to model its thermal interactions with its environment, and a third used to model its interaction with its source of power.

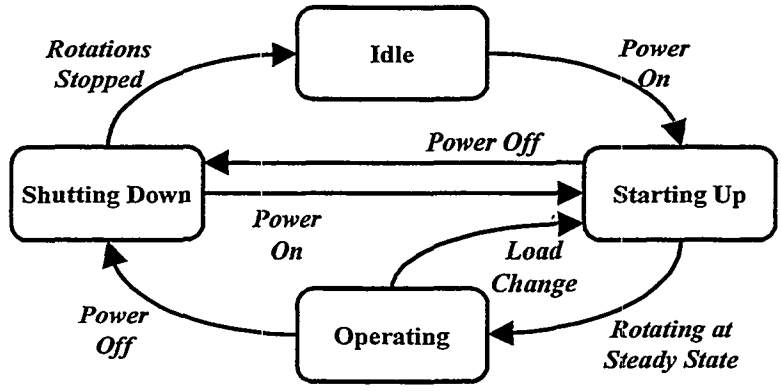


Figure 4-6. A State Transition Diagram

To describe how the functional block responds to inputs while in a given operating state, the analyst uses a data flow diagram (Figure 4-7) combined with one or more truth tables (Figure 4-8). As shown in the figure, the diagram consists of four elements. The circles represent processes that transform flows. The lines represent the interconnections between processes and carry the various flows, which are identified by labels. The flows themselves are described as collections of attributes. The horizontal lines represent a "store" which contains the attributes that document the current state of the functional block (e.g., a functional block representing a chemical process might contain a store with attributes that specify the current composition, temperature, quantity, etc. of the chemicals in the process).

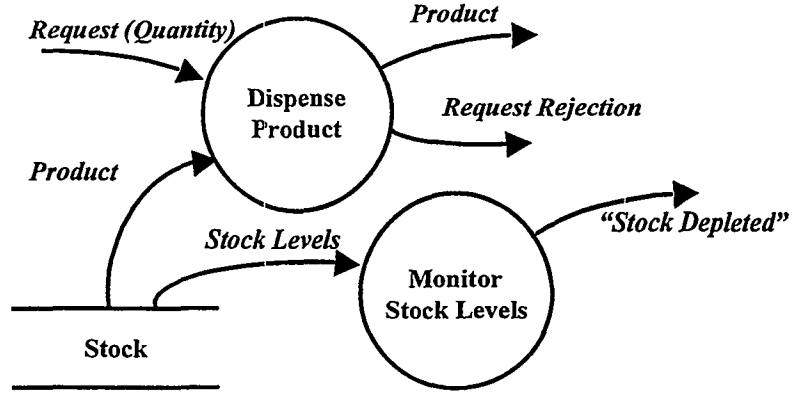


Figure 4-7. A Data Flow Diagram

Number of Items in Stock	—	Number of Items Requested	Number of Items Produced	"Request Rejection"
+			Number Requested	F
0			Number Requested	F
-			0	T

Figure 4-8. A Truth Table

In implementing the truth tables, the central idea is to specify what values a given attribute assumes when the input attributes on which the attribute depends assume various combinations of values. It is a straightforward task to develop this specification when the attributes all assume a finite set of values. When the attributes assume a continuous range of values, the task is a little trickier. To address this issue, the research team borrowed the notion of "landmark values" from the qualitative physics community. In this approach to modeling attribute values, the goal is to identify key points in the attribute's range of values at which behavior somehow changes. For example, if the attribute under consideration is the amount of fluid in a storage tank, then it is reasonable to assume that the tank might exhibit three different behaviors corresponding to the tank's being empty, to the tank having some amount of liquid in it, and to the tank's being full. Given this, two landmark values would be assigned to the tank (i.e., "tank empty" and "tank full").

In most cases, a given truth table will have a single output column that corresponds to a single attribute on one of the output flows from a process and some number of input columns (corresponding to some or all of the attributes on the input flows to a process). In some cases (e.g., when the process involves a "test"), attribute flows are combined into a single column, as shown in Figure 4-8. Also, as shown in this figure, multiple tables may be combined for the sake of compactness if the logic of the tables being combined is sufficiently simple.

4.2.4 An Example of Modeling Functionality

To illustrate the approach to modeling described in this section, consider the following example. A system being analyzed receives a series of messages from some external source and stores them in an internal buffer until told by the some other entity to forward the messages to that entity. The system is also to monitor the incoming messages for a certain "signature" (i.e., a distinctive set of attributes in a message) and to automatically forward messages matching this signature to a third entity upon detection. The top-level system structure diagram for this example is shown in Figure 4-9. Figure 4-10 shows the functional decomposition of the "smart buffering" block.

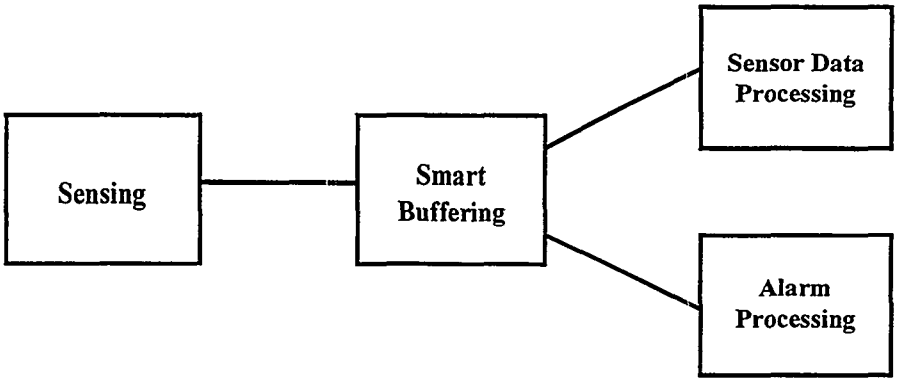


Figure 4-9. The Top-level System Structure Diagram

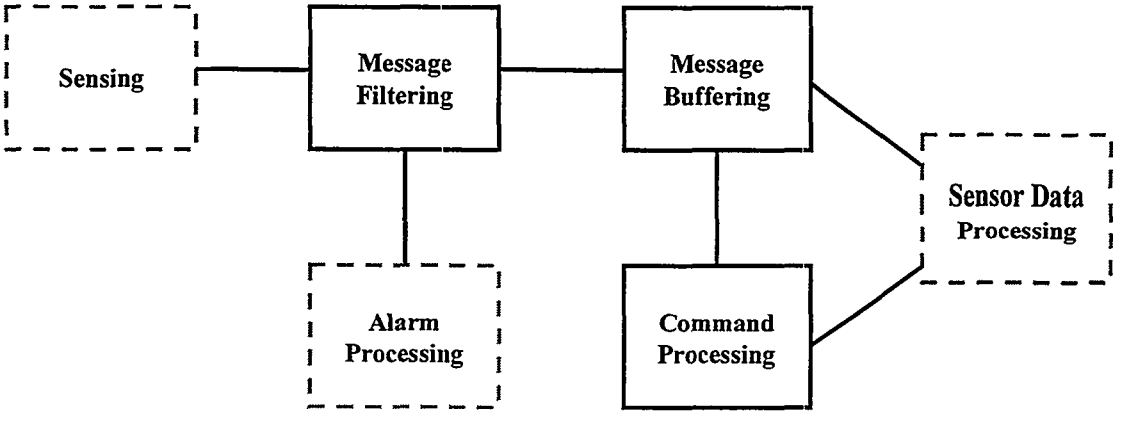


Figure 4-10. The System Structure Diagram for "Smart Buffering"

Figure 4-11 is the interaction diagram that corresponds to Figure 4-9. Figure 4-12 shows the interactions within the "smart buffering" functional block. Note that the flow "smart buffer responses," decomposes into two flows ("sensor messages" and "empty buffer messages") in the lower-level diagram.

Figure 4-13 is the state transition diagram for the "message filtering" block of Figure 4-12. Normally this block is in the "idle" state, waiting for a message to arrive from the "sensing" block. When a message arrives, the "message filtering" block moves to the "evaluating message" state, generates either a standard "sensor message" or an "alarm message," and then transitions back to the "idle" state.

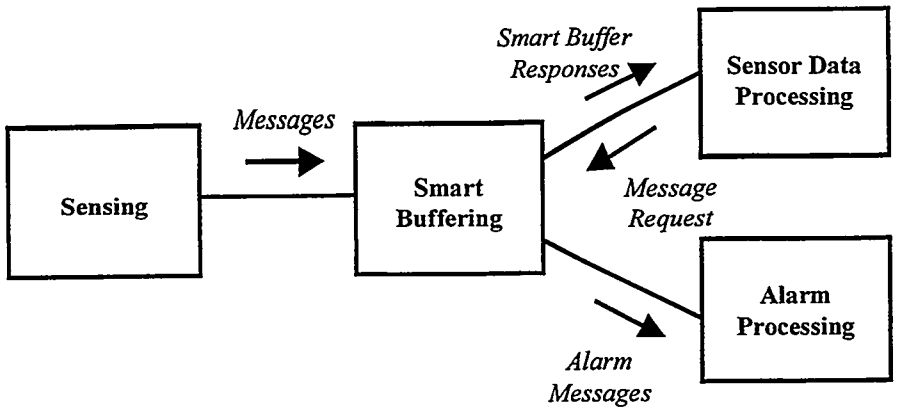


Figure 4-11. The Top-level Interaction Diagram

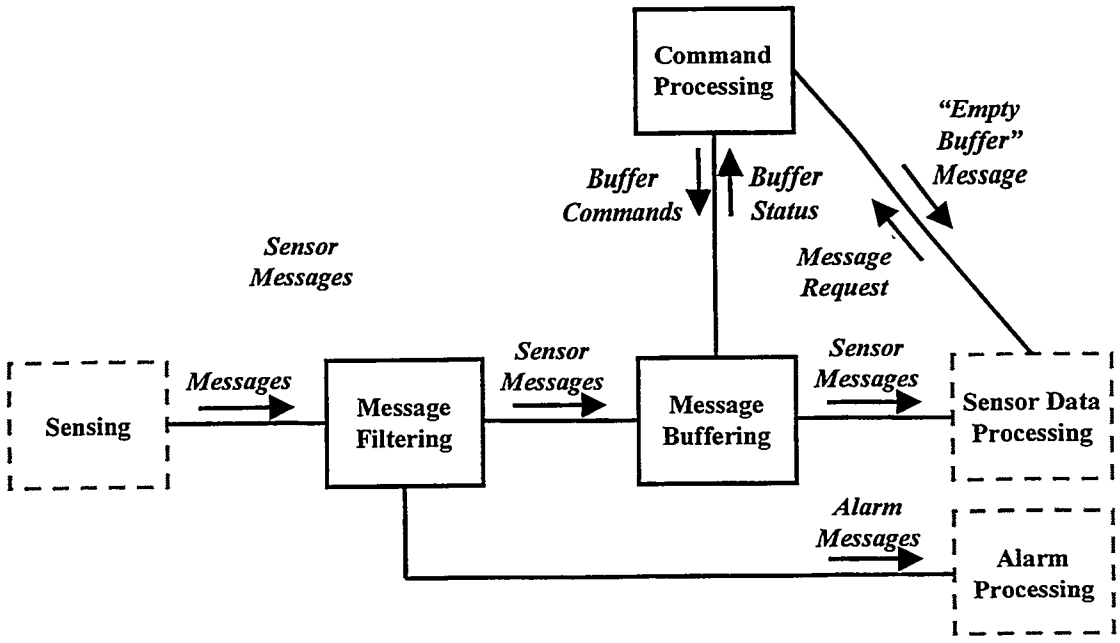


Figure 4-12. The Interaction Diagram for "Smart Buffering"

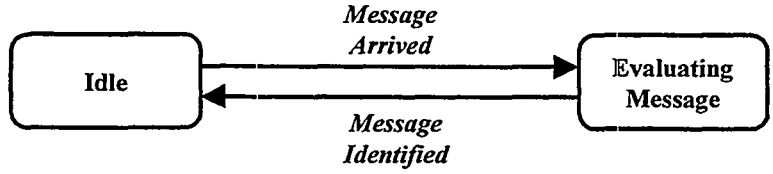


Figure 4-13. The State Transition Diagram for "Message Filtering"

Figure 4-14 is the state transition diagram for the "message buffering" block. As in Figure 4–13, this block spends most of its time in the "idle" state. When a sensor message is received, the block moves to the "storing message" state and then returns to "idle" once the message is stored. If the block receives a "buffer status query," it moves to the "processing query" state, where it counts the number of messages currently stored, transmits this number to the "command processing" block, and then returns to "idle." If the block receives a "send message" command, the block transmits the oldest message in its queue to the "sensor data processing" block, removes this message from the queue, and then returns to "idle".

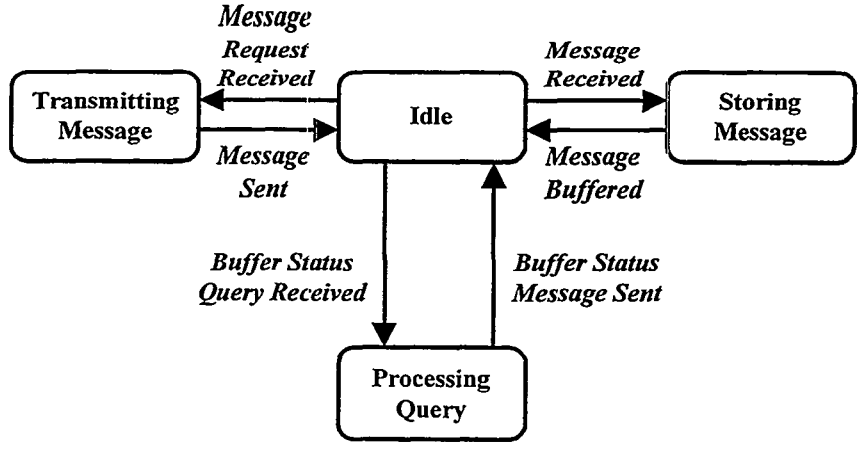


Figure 4-14. The State Transition Diagram for "Message Buffering"

Figure 4-15 is the state transition diagram for the "command processing" block. While in the "idle" state, this block waits for "message requests." When one is received, the block queries the "message buffering" block to determine if any messages are currently queued up. If they are, then the "command processing" block sends a message to the "message buffering" block instructing it to forward a message to the "sensor data processing" block. If no messages are currently queued up, then the "command processing" block returns an "empty buffer" message to the "sensor data processing" block.

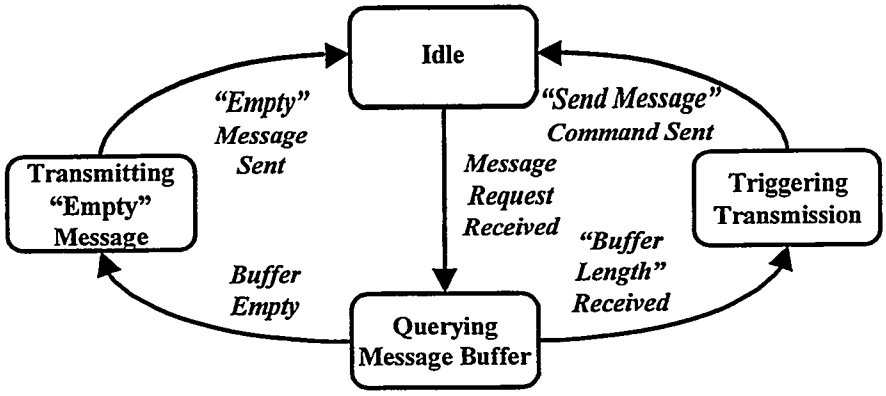


Figure 4-15. The State Transition Diagram for "Command Processing"

At this point, the "smart buffering" model contains ten distinct states. Of these ten, the three "idle" states perform no actions and therefore have no associated data flow diagrams. The data flow diagrams for the rest are found in Figure 4-16 through Figure 4-22.

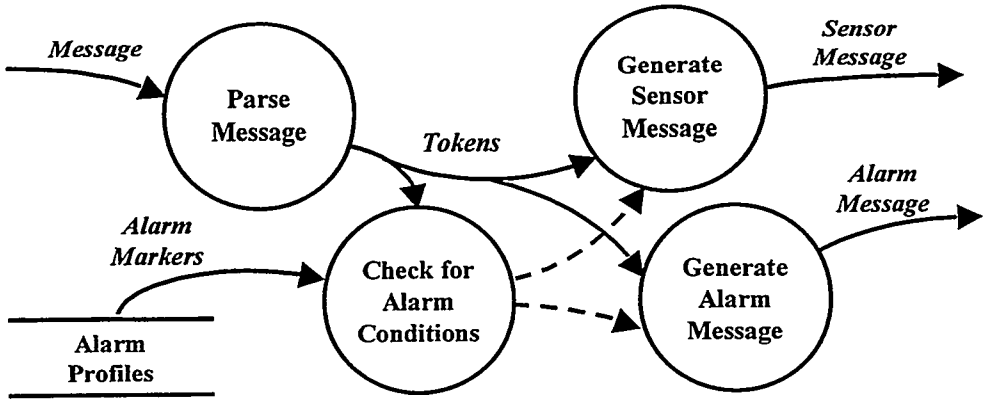


Figure 4-16. Data Flow Diagram for "Evaluating Message"

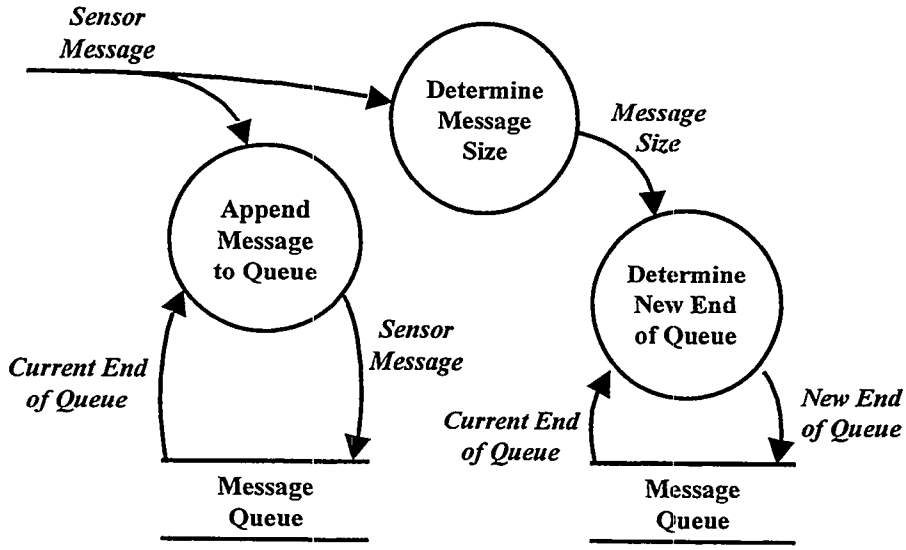


Figure 4-17. Data Flow Diagram for "Storing Message"

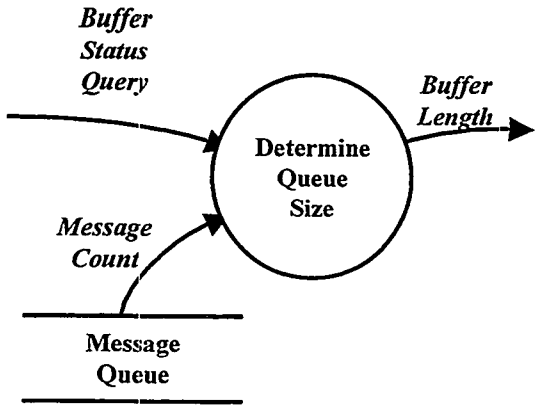


Figure 4-18. Data Flow Diagram for "Processing Query"

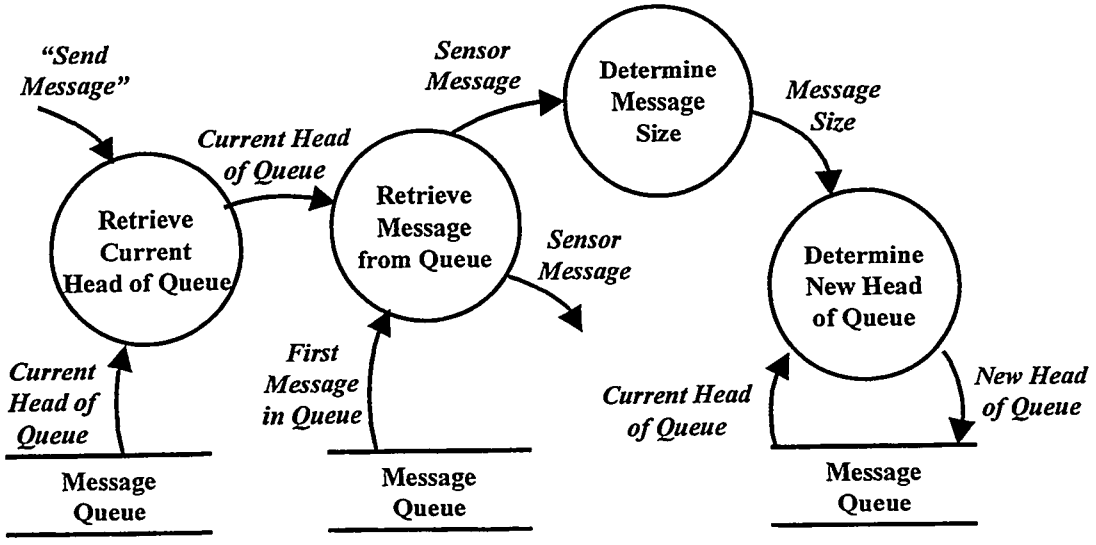


Figure 4-19. Data Flow Diagram for "Transmitting Message"

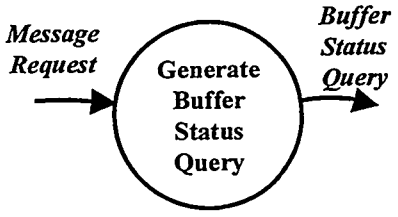


Figure 4-20. Data Flow Diagram for "Querying Message Buffer"

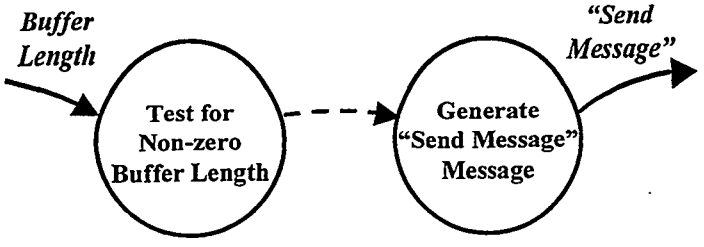


Figure 4-21. Data Flow Diagram for "Triggering Transmission"

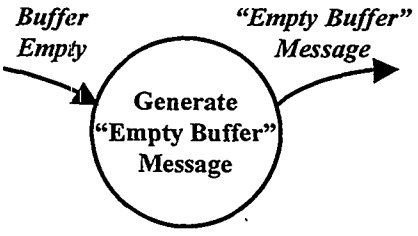


Figure 4-22. Data Flow Diagram for "Transmitting 'Empty' Message"

At this point in the modeling of the example, the analyst has created the full graph for the "smart buffering" block. All that is left is to supply the truth tables that bridge process inputs and outputs. For the sake of brevity, only one of these will be supplied (as shown in Figure 4-23). Before leaving the example, there are several points worth noting. Chapter 5 will explain how an analysis algorithm traces a graph created in the way just shown. While the main focus of this tracing will be the traversal processes, it is possible that causal links also run through the "stores" contained in data flow diagrams. Said differently, since the state of an object's internal attributes can affect that object's outputs, it is possible for one interaction to set the attributes of an object so that later transactions are affected in a way that is of interest to the analyst. This phenomenon is at the heart of life cycle modeling (as discussed later). Second, as can be seen from this example, even relatively simple models can produce a significant number of views. For this reason, doing this work by hand can be tedious. At the same time, automation support is invaluable and simplifies much of the process by performing many of the bookkeeping functions needed to ensure consistency among model views. Third, even if the modeling is not done to the point that assessment models can be automatically derived from the system model, the exercise is still worthwhile because the system model provides the analyst with a framework that encourages a structured evaluation of the system under consideration.

Buffer Length	Control Flow
Zero	Does Not Fire
Nonzero	Fires

Figure 4-23. Truth Table "Test for Nonzero Buffer Length"

4.3 Documenting Abnormal Component Behaviors

Given the normal system model developed as described above, the analyst begins the process of considering how things can go wrong in this model. In general, problems fall into two categories: problems with communications and problems with functional mechanisms. An example of the first would be the inclusion of new message types under the flow labeled "buffer commands" in Figure 4-12. As result of this, Figure 4-14 would have to be modified as shown in Figure 4-24 to accommodate the possibility of erroneous messages. Related to this is the corruption of "store" contents in a data flow diagram. For example, if the "alarm markers" in Figure 4-16, were altered, then the detection algorithm might fail to distinguish normal sensor messages from those that constitute alarms. An adversary might use this fact either to thwart detection or to over-sensitize the system so that it generated alarms when no alarm conditions existed. An example of the second type of problem is shown in Figure 4-25. Here, the state transition diagram shown in Figure 4-14 is updated to account for the fact that storage mechanisms are always of finite capacity.

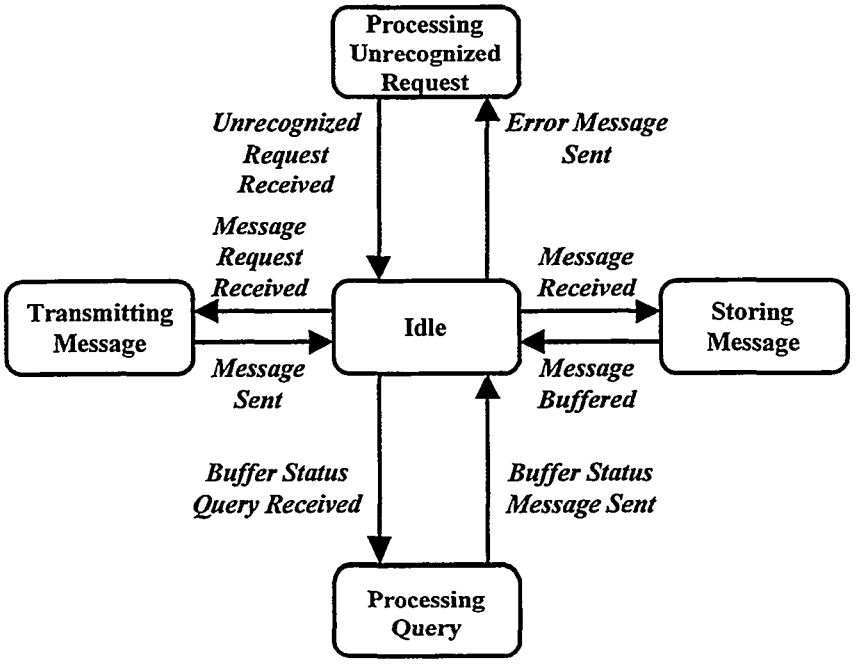


Figure 4-24. Updated State Transition Diagram for "Message Buffering"

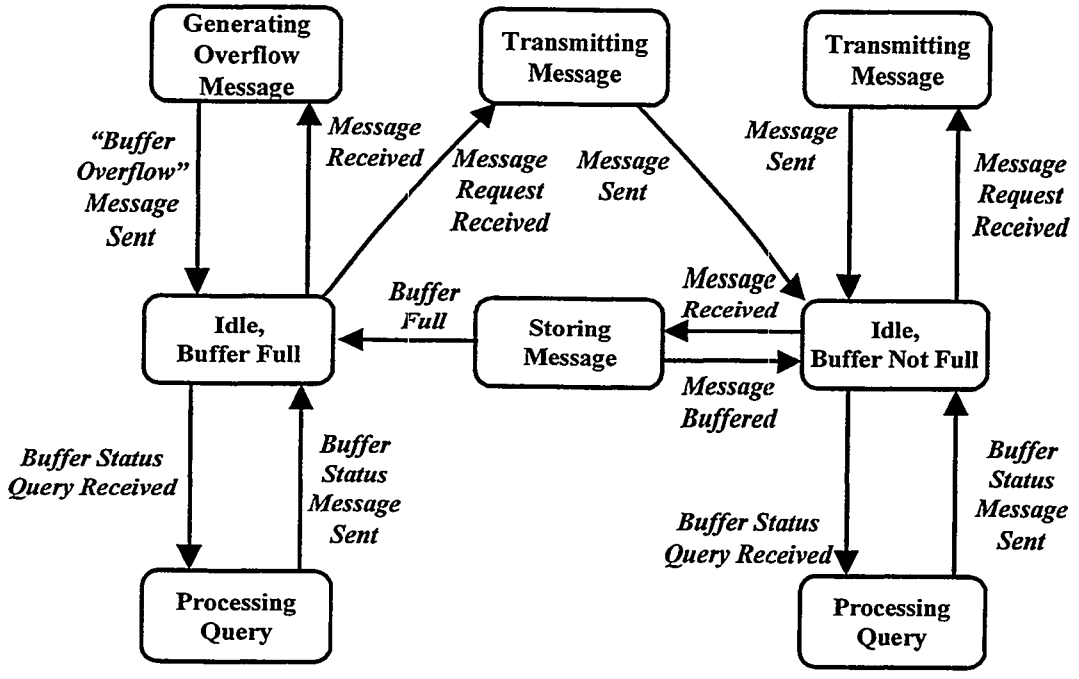


Figure 4-25. Another State Transition Diagram Variation for "Message Buffering"

4.4 Documenting the "Physical" Model

As described so far, nothing that the analyst has done documents how the functional blocks are aggregated into subsystems to be implemented. Given Figure 4-10, a system designer might choose to divide the system as shown in Figure 4-26 or as shown in Figure 4-27. This sort of partitioning would correspond to the kind of work done by a system architect in the early design phases of a project.

As the project proceeds and decisions are made regarding the specific ways in which functional blocks will be rendered (e.g., "these will be hardware, these software, and these functions will be performed by human operators"), the effect, from the viewpoint of the system model, is to add additional functional blocks to the model. For example, suppose that some of the blocks in the system were to be rendered as digital hardware. To the application-specific functionality already described in the system model, the analyst would add those blocks that describe the functionality of the digital hardware, independent of the specific tasks that the hardware is intended to accomplish. These could include how the hardware relates to its power source, how it interacts with its thermal environment, and how it behaves from a radio frequency viewpoint (i.e., whether it radiates electromagnetic waves or is vulnerable to this sort of radiation). One way of thinking about this is to consider that any given physical component is described by a set of concurrent models (Figure 4-28). These models link both to models contained in other external components and to one another within the scope of the component that they represent. Given this, the functional blocks allocated to a given component can then be

viewed as nothing more than additional models operating in parallel with and connected to the models that are already a part of the physical component (Figure 4-29).

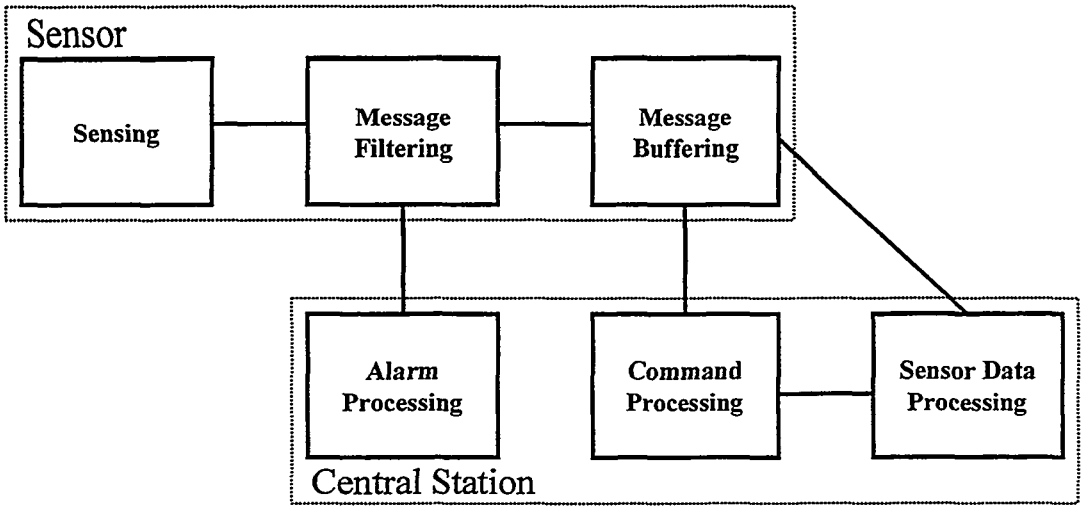


Figure 4-26. One Partitioning of Figure 4-10

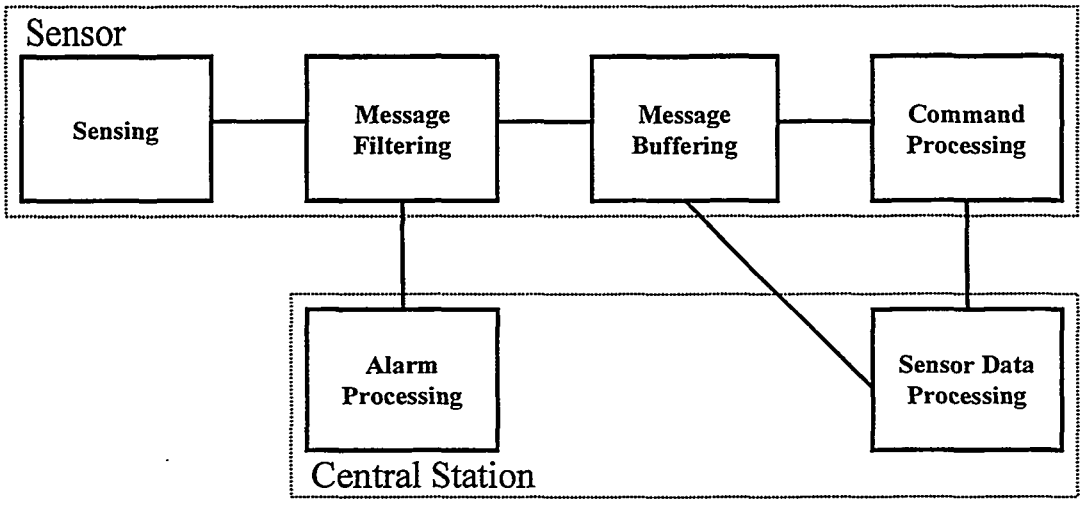


Figure 4-27. An Alternative Partitioning of Figure 4-10

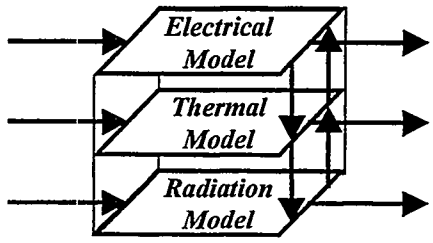


Figure 4-28. A Physical Component in the System Model

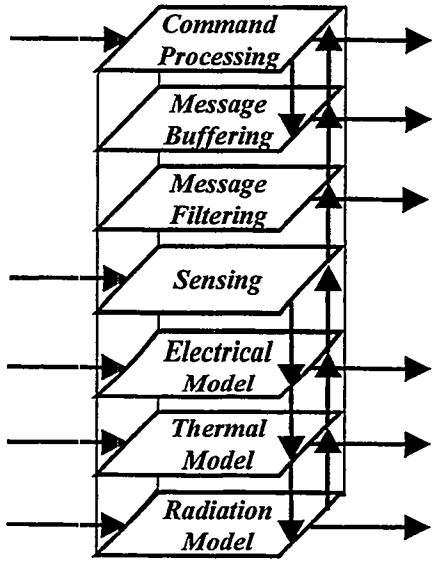


Figure 4-29. The Physical Model of the Sensor Component in the System Model

4.5 Documenting the Spatial Aspects of the System Model

Just as the last section described "physical" components from a logical perspective by using the modeling constructs described in the start of this chapter, an analyst can use these same constructs to model spatial regions in a system being assessed. Suppose, for instance, that the system being assessed resides within a building that is located in the middle of a fenced area. The model for this facility might be as shown in Figure 4-30. As with the system structure diagrams shown earlier, each of the functional blocks shown in this diagram can be hierarchically decomposed. Flows can also be added to the connections to show what moves within the facility and to describe how the facility responds to these flows.

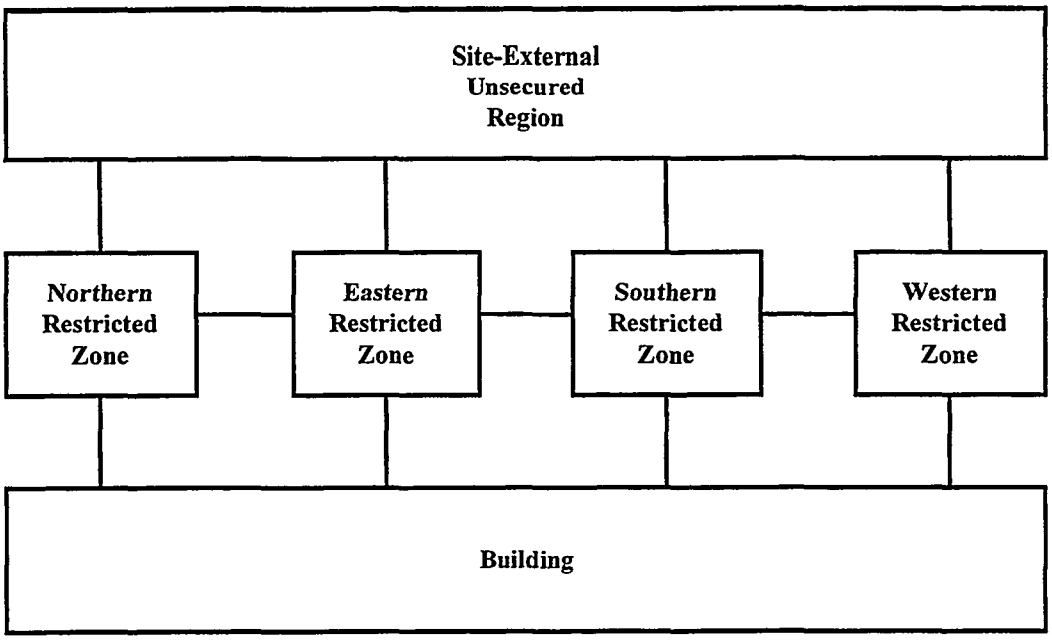


Figure 4-30. Top-level System Structure Diagram of the Secure Facility

Once the analyst completes this part of the model, mappings between the physical models and the spatial models are established. These relationships generally take two forms. In the first, the "functional blocks" that populate the physical model deliver "flows" to the physical components contained in the regions represented by these blocks. For instance, a physical model block representing a room could deliver an ambient temperature to the hardware and people located in that room. In the second, the blocks in the two domains deliver other "blocks" to one another. An example of this would include a secured room presenting a computer to an "insider," who then carries on a set of interactions with the computer with the goal of subverting the computer in some way.

4.6 Documenting Component Life Cycles

Because the way in which a system component behaves in a given interaction can depend on what has happened to the system before to this interaction, in some circumstances it is important to document some part or all of the component's life cycle. The primary mechanism for this is a "life cycle diagram," like that depicted in Figure 4-31. In this diagram, each of the ellipses corresponds to a system structure diagram that documents the relationships in which the associated component participates at that point in its life cycle. Because the component that is the focus of the diagram has a "state," it is possible for a component associated with this component in one life cycle stage to influence another component associated with the main component at a later life cycle stage. It should be noted that this state is not only those things represented by the component's internal attributes but also the functionality that the component brings to bear in any

situation. Either is subject to change in one stage in such a way that subsequent stages are affected.

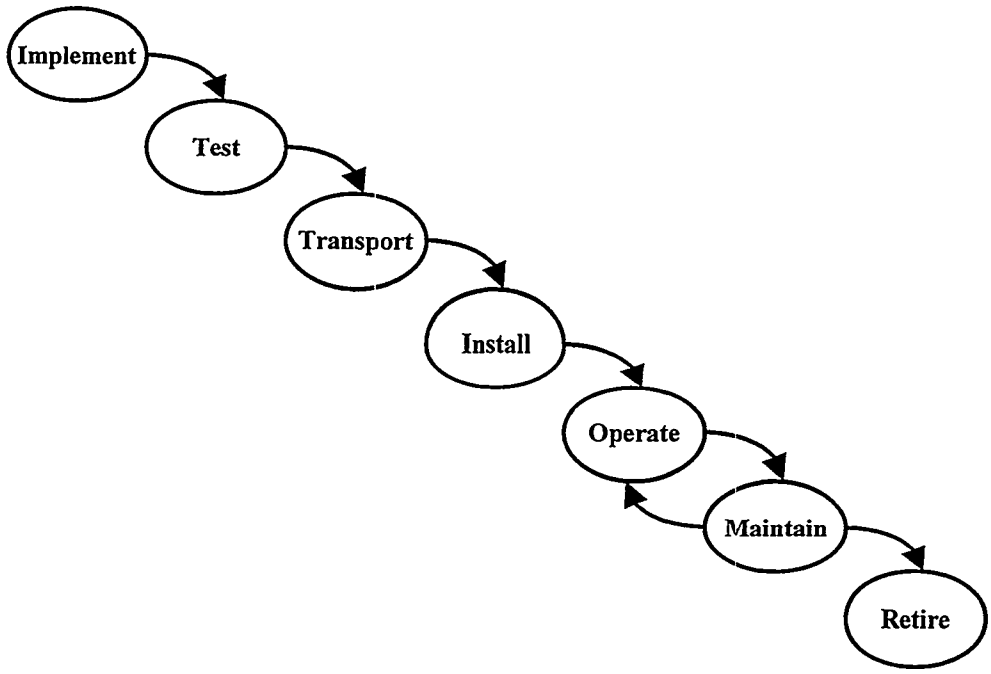


Figure 4-31. The State Transition Diagram for "Command Processing"

In creating this sort of diagram, it is important to distinguish between the product that is being modeled and the elements used to create the product. For instance, a piece of software and the design used to create the software are not the same thing. The first is derived from the second, but the two are only related by means of a series of translation steps, each of which may need to be understood in order for the system to be fully assessed.

4.7 Conclusion

The goal of developing an explicit system model is to identify what is understood about the various dependencies that exist in the system being assessed. This includes documenting how the system behaves, describing which mechanisms deliver that functionality, identifying where in space the components exist in relationship to the environment and to each other, and capturing the relationships in which each system component takes part. By documenting these dependencies, the analyst knits together a graph that can explain the full set of cause-and-effect relationships within a system. Given this graph, the analyst then can assess whether specific sets of events lead to undesirable results or determine how certain undesirable events can be realized. This

leads to a more complete assessment than is possible with the standard checklist-based techniques in use today.

5 Surety Evaluation Using Object Models

For the surety analyst, the object model is not an end in itself, but a means by which one systematically embodies their understanding of the system. This embodiment is useful because it can be queried, searched, and probed by surety analysts as they search for possible sources of vulnerability, unreliability, or risk. Therefore the purpose of this chapter is to describe how an object model – constructed using methods from the previous chapter – can be used to perform such a surety analysis.

Surety analysts use a wide variety of tools to assess systems, but these tools can be broadly classified as inductive, deductive, and simulation models. This chapter will examine examples from each of these classes and describe rules that can be used to extract either the models themselves or the information that they produce from the object model described in Chapter 4.

5.1 Inductive Surety Models

Inductive surety analysis models have the characteristic that one postulates a particular set of system boundary conditions and then logically examines the system to determine how it will respond to those conditions. These inductive models differ from simulation models in that simulation models are generally based on what one might think of as detailed "basic science" models, while inductive surety models are generally based on *logical* conditions and constructions. Thus, inductive surety models are often higher-level, fast-running models that can be thought of as extrapolating from and interpolating among the results of a number of simulation calculations. Inductive models are important because they allow the surety analyst to consider a much wider array of scenarios than would be computationally feasible using a simulation-based approach.

Two common inductive models – failure modes and effects analysis, and event tree analysis – and how they can be obtained from the object model will be examined here. These methods themselves were described briefly in Chapter 2. The patterns used to extract the models for these two methods from the object models should provide insight into how other inductive methods can make use of object models as well.

5.1.1 Failure Modes and Effects Analysis

Recall that the purpose of a FMEA[1] is to examine the individual components and assess the effects of their failure on their systems and on other systems and subsystems. To accomplish this, the analyst goes through the components of the system one by one, and for each component considers every known failure mode individually. An automatic processor can accomplish this task if one constructs an object model using the methods described in Chapter 4.

The object model must contain two classes of information in order to automatically produce a FMEA. First, it must be able to distinguish whether particular combinations of

conditions and attribute values represent normal or failure conditions. Second, it must contain a method for determining whether a particular condition of the system is acceptable, undesired, or unacceptable. Given this information, it is a simple task to select the individual failure conditions one at a time, and propagate the effects of that failure through the rest of the object model. The model can then determine whether the resulting system condition is acceptable. Furthermore, if the consequences of being in a particular system condition can be quantified, and the likelihood of each failure condition is known, the automated processor can easily determine the information necessary to extend the FMEA into a failure modes, effects, and criticality analysis (FMECA). Thus, to automate the performance of a FMECA, the object model must be able to (1) propagate the effects of an arbitrary event or change in attribute value through a model to determine how it affects the state of the system, (2) select all known failure modes for all components of the system one by one for propagation through the object model, (3) categorize the resultant system conditions to determine their acceptability and/or consequences, and (4) combine this information with a likelihood for the original failure so that appropriate risk descriptors can be assigned to each.

The main characteristics that are required of the object model to perform a failure modes and effects analysis are the ability to propagate the condition of the system from a failure occurrence to the ultimate state of the system, and the ability to distinguish between the normal and failure modes for individual components. The portion of this analysis that cannot be automated, however, may include documenting the method by which a failure would be detected in the operating system and suggesting and documenting possible actions to reduce the failure rate or effects. Indeed, possible mitigators for component failure are generally selected through the use of imaginative thinking that is informed by the ranked risk results from this analysis. Since these mitigators will likely modify the system configuration, they will require modifications to the system object model and additional application of FMEA techniques to ensure that the mitigators themselves do not introduce new failure modes.

The FMEA's close cousin, the hazards and operability analysis[2], can also be automatically evaluated from an object model. Recall that the HAZOP method focuses on qualitative deviations of key system operating parameters from their normal, nominal, or design values. These operating parameters are embodied within the object model in the form of discrete attribute values. Thus, an automatic processor that was aware of nominal vs. off-nominal values for discrete attribute values could systematically vary those values one at a time in much the same manner as component failure modes were considered for the FMEA analysis. This would enable the processor to link the causes of consequences with their effects. What is not possible without additional logic is HAZOP's additional requirement to use deductive logic to obtain probable causes for the parameter deviations. This could be done using the method of fault tree analysis, which is described later in this chapter. This can, however, automate large portions of the HAZOP analysis through the use of an object model.

The above description provides an overview of the method by which one could extract either a HAZOP analysis or a failure modes and effects analysis from a common object

model. Appendix A contains a more detailed description of the rules required to perform this extraction.

5.1.2 Event Tree Analysis

A second inductive risk assessment technique that can be automated using an object model is event tree analysis.[3] While FMEA considers a system with multiple sets of initial conditions (based on individual component failures), an event tree begins with a single initial condition for the system and examines the spectrum of possible outcomes given that particular initial condition. Event tree construction begins with the selection of the initial condition that must be propagated through the object model. However, unlike a FMEA, which produces only a single system end state, an event tree produces many system end states. The automated processor, as it traverses the object model, will come to a number of subsystems, components, and/or states that may be entered randomly, or may be due to the effect of a component failure or external influence on the system. The FMEA processor assumes that the condition of each subsystem and component remains as it was initially unless forced to change as a result of the component's failure. The event tree processor, however, views each such potential change as a branch point in the event tree model. Thus, it must remember the state of the system at that branch point and choose one of the possible options to propagate through to a final system state. It must then go back and pick up the saved system state, choose a different option, and propagate that through to a final system state. Since there may be several such possible branch points, this process is recursive and is repeated until all such options are either exercised or eliminated (on physical or probabilistic grounds). Each unique combination of branch point choices and its associated final system state can be thought of as a path through a traditional event tree or, alternatively, a system scenario.

An analyst may wish to select different types of branch points for different analyses, depending on the eventual use of the analysis results. Three typical ways to determine branch points are:

- specific points within the object model as defined only by the analyst (possibly including the effects of operator actions or external phenomena operating upon the system)
- random events that are incorporated within the object model (mainly component failures) or
- all points in the object model at which multiple discrete attribute values are physically plausible

Obviously the last of these methods may result in an extremely large event tree model that will produce huge numbers of scenarios. The object model must be capable of representing all three of these types of branch points, and must provide a method by which the analyst can select one of these for a particular event tree analysis.

Note that in order for an event tree analysis to be able to rank these scenarios in a manner that is useful to the surety analyst, the acceptability and/or consequence of the resultant system state must be determined for each scenario. In addition, a traditional event tree analysis determines the likelihood of each such scenario by multiplying together the conditional probabilities associated with each of the branch points in the scenario. If likelihoods are included in the object model for all of the object model events that can become branch points, the automated event tree processor can easily perform a similar computation for each scenario produced from the object model. If some of these likelihoods are not incorporated in the object model, the scenario likelihood computations can be carried out using the available data and then grouped according to the branches chosen at those branch points for which likelihoods were not available (i.e., one could perform assessments that are conditional upon a particular branch being chosen for these unquantified branch points).

It is not unusual for event tree analyses to produce many thousands of possible scenarios given a particular set of initial conditions. An analyst is generally unable to examine such a large number of scenarios in the appropriate level of detail. Therefore, event tree analysts often employ a method known as "binning." In order to bin event tree scenarios, an analyst decides which characteristics are important over all of the scenarios. These characteristics are generally related to the consequences that will accrue to the system due to this scenario. Once these characteristics are specified to the automated processor, the processor then groups all scenarios that contain similar characteristics so that the analyst can look for patterns among those scenarios. In essence, the analyst defines a number of "bins," each of which is defined so that the important characteristics of all scenarios that fit that bin are similar. Using this process, the analyst can examine perhaps a few dozens of bins instead of many thousands of individual scenarios.

The above description provides an overview of the method by which one could extract an event tree analysis model from a common object model. A more detailed description of the rules required to perform this extraction can be found in Appendix A.

5.2 Deductive Surety Models

While inductive analysis models determine the response of the system to particular boundary conditions, deductive models begin by defining a condition or consequences to be avoided. The analyst then uses deductive reasoning to determine how this event can occur. The most commonly used deductive analysis method is fault tree analysis. While deductive methods such as fault tree analysis are not as closely related to simulation models as are their inductive model counterparts, it is still critically important that the fault tree itself reflect a thorough understanding of the basic science involved in the system. However, while inductive methods seek to embody this basic science in a direct manner, its application in deductive methods must be embodied in the logical conditions and constructions that go into the fault tree or other model.

Since fault tree analysis is the most commonly used form of deductive model, this section will concentrate on methods for the automatic derivation and solution of fault trees from

the object models described in Chapter 4. We will also describe briefly the application of the object model to influence diagrams and reliability block diagrams.

5.2.1 Fault Tree Analysis

The objective of fault tree analysis is to provide a formalized method to assist the analyst in reasoning through the causes for particular failures, and in documenting the steps involved in that reasoning for the benefit of others. The basic premise of fault tree analysis[4] is that

- *If* the analyst takes very small logical steps, and
- *If* the analysis is logically complete at each of those small steps,
- *Then* one can be reasonably confident that the overall logical model is a complete representation of the modes by which the system may fail.

In other words, one builds a fault tree by ensuring completeness at each small logical step and allows the final assembly of the model (often by software) to bring out both the large-scale and small-scale interactions among the parts of the system. The formal name for this technique of exhaustive, small logical steps is "immediate cause." A similar concept exists in the world of object modeling: the object model must itself be a detailed and complete representation of how the system works if one is to have confidence that the model is an accurate representation of a system's true behavior. Errors and omissions typically come about because of a lack of rigor or the presence of unstated assumptions.

The object model embodies the detailed flow of data, commodities, and stimuli through the system as a series of state machines and flow diagrams. The construction of a fault tree from such a model requires four conceptual steps:

1. Specifying the initial conditions for the system to be analyzed
2. Specifying the condition that is to be avoided or ensured in terms of the states and attribute values found in the object model
3. Tracing backward through the state machines and data flow diagrams to determine the immediate cause of each such condition (this process occurs repeatedly in a recursive manner)
4. Terminating the recursive tracing when all fundamental causes are identified

While step four may seem so obvious as to be discounted, it turns out to be one of the more difficult parts of the fault tree extraction method. This is because, depending on the purpose and the boundaries of the analysis, that which is a fundamental cause in one analysis may require much further elaboration in a different analysis. In yet another analysis, this same fundamental cause may occur at a level of detail that overwhelms the analyst's capacity to understand it. In a traditional fault tree analysis, decisions about the

analyst's capacity to understand it. In a traditional fault tree analysis, decisions about the level of detail and termination are made in the mind of the risk analyst (faulty decisions in this area often lead to fault tree analyses that are incomplete or otherwise deficient). The challenge is to find an automated termination method that provides enhanced flexibility for the analyst without falling victim to these same issues of incompleteness.

It is easy to trace backward through the data flow diagram (or, as we are using it, the process flow diagram) to determine the immediate, necessary, and sufficient conditions required to produce any set of conditions within that diagram. One simply looks at the output of the diagram in terms of the process that created that output (this output will be created by a truth table within the model). The truth table presents the complete set of conditions that can immediately cause the required output. These conditions can be a function of other processes within the flow diagram, processes in other objects, and events from the state transition diagram of either this object or other objects in the model. Since the flow diagram shows the source of the flows that are from within this object, one need only trace backward through this causal graph to determine the causes of this intermediate condition. If the flow can originate from outside of this object, then one must take the additional step of searching the other objects in the model to determine all possible origins of the flow. This same process can now be repeated recursively to define the causes of these new intermediate conditions until the terminating conditions described above are satisfied. This is key: the object model, and especially the data or process flow diagram, acts as a deterministic causal graph that can be examined through automated qualitative reasoning techniques to build a fault tree model based on any point in the object model. The ability to trace backward through the flow diagrams (which is made possible by the truth table logic) enables this to occur.

The deterministic nature of this model suits many systems well, although it obviously cannot handle stochastic or nondeterministic systems very well. Nondeterminism can be an important factor in the operation of many computer systems. Additional research will be required before fault tree analysis methods can be applied to such systems – either by traditional risk assessment methods, or by the object-oriented methods described in this report.

The description given here provides an overview of the method by which one could extract a fault tree analysis model from a common object model. Appendix A contains a more detailed description of the rules required to perform this extraction.

5.2.2 Influence Diagrams

While influence diagrams[5] form a different and unique technique in the risk and reliability analysis community, they are not really unique with respect to the common object model. Recall that one of the principal advantages of the influence diagram method for the analyst was the ability to simultaneously reason inductively and deductively in a single model. This enables the analyst to initially state the objectives of the analysis (much as one would do in a deductive fault tree model), as well as some of the events and conditions that are expected to influence the system (similar to initial conditions in an inductive event tree model). Using combinations of the inductive and

deductive logic enables the analyst to develop the connections between them and to find additional, possibly unexpected, influences on the system.

While this combination of inductive and deductive reasoning is a strong advantage for a human analyst, it does not provide benefits when one is extracting information from an existing object model because that model, if properly constructed, has already identified the expected and unexpected influences on the system. Therefore, the analyst will be starting either with a set of initial conditions and seeking to inductively identify how they might progress through the system (an event tree analysis), or with a set of resultant conditions and seeking to deductively identify how those conditions might come into existence (a fault tree analysis). If one were concerned about how particular input conditions might result in particular output conditions, one could specify a fault tree analysis and adjust the terminating conditions for the search to appropriately highlight the input conditions of concern. For this reason, it would be of marginal utility to develop a separate facility for producing an influence diagram model, given that the system is already capable of producing both fault tree analysis and event tree analysis models. Thus, while the research team did not seek to identify rules to produce such a model, it may be beneficial in future research to construct such rules either as an academic exercise, or, if justified by a future analysis need, for actual production use.

5.3 Discrete-Event Simulations

Previous sections in this chapter described how one would extract some of the traditional risk and reliability analysis logic models from the common object model. While these logic models can be very valuable in surety science, there is also a strong need to use simple simulation techniques to help ensure the surety of the system. Since the aim of the common object model is to embody all of the behavioral information about a system, it is relatively simple to use that behavioral information as the basis for a simulation methodology. Note that because of the requirement to extract logic models, we embodied the system behavior in the object model in terms of only discrete state and attribute values. For this reason, systems that must be described in terms of continuous variables will be more difficult to simulate using this common object model than systems that naturally operate in the discrete realm.

As with most object-oriented modeling methodologies, the method for performing a single simulation based on the object model is very simple.[6] One specifies the initial and boundary conditions of the system in terms of the states and attribute values embodied in the common object model. This may, for example, be the equilibrium state of the system before it is perturbed by some outside influence. If the initial and boundary conditions represent a nonequilibrium system state, one simply propagates these conditions through the object model using rules established for that particular object modeling methodology to determine the response of the system to that set of initial and boundary conditions. If the initial conditions represent an equilibrium state, one typically applies the perturbation to the system at an appropriate point in time, and then, as before, propagates these conditions through the object model using the rules of the object modeling methodology to determine the system response. It is also possible, in time-dependent analyses, to

apply additional perturbations to the system at later points in time to determine how these affect the overall performance of the system. In all of these cases, the result is a description of how the system performs given the stimuli that are applied to it. If events within the object model are probabilistic in nature, the analyst must determine whether particular outcomes for these events are to be selected a priori, or whether multiple simulations are to be run to take into account these possible alternative outcomes. If multiple simulations are run, the resultant multipath simulation takes on characteristics of an event analysis, as described previously.

Discrete-event simulations are valuable to the surety analyst because they provide answers to very specific questions regarding the behavior of a system under specific conditions that may be of pressing interest. This is in contrast to the event tree analysis methodology, which seeks to understand the universe of possible outcomes given a set of initial conditions. The discrete-event simulation methodology provides rapid (and perhaps more detailed) assessment of a system's specific behavior under specific circumstances, while the event tree analysis methodology requires much more computational effort and produces a statistical representation of the variety of possible scenarios that may occur. Both of these perspectives are very useful to surety analysts as they seek to achieve a complete understanding of the cause-and-effect relationships that drive the performance of a system.

5.4 Other Methods

A number of other analysis methods can be supported to varying degrees by the common object modeling methodology described in this report. For example, while it was our intention to embody location information in the common object model, we were not able to fully achieve that before the scheduled end of this project. However, it is still possible to support location-based analyses using information derived from the common object model in the following manner. One can extract a fault tree or other deductive logic model from the common object model, and solve it using traditional analysis techniques to obtain the resultant minimal cut sets.* Then, using techniques developed for vital area analysis,[7] one can associate the events in the cut sets with locations where a person could deliberately accomplish the noted failure mode. Once this association is made, the cut sets can be further processed and reduced to produce location-based cut sets. Each location-based cut set contains one minimal set of locations to which a person would require access in order to cause the undesired event for which the system was analyzed. This information by itself is valuable to security practitioners because it can be subjected to importance analysis to determine those locations that are the most important to protect. In addition, if the list of location-based cut sets is further processed to obtain its mathematical dual, one obtains what are called "protection sets." Each protection set

* Briefly stated, an individual minimal cut set represents one set of events and component failures which, were they all to occur at an appropriate time, would cause the undesired event for which the system is being analyzed. The entire list of minimal cut sets represents the total catalog of ways by which the system can be forced into this undesired state given the behavioral information embodied in the common object model.

represents one combination of locations which, if all were completely protected, would result in a system that had no vulnerability to the particular undesired event for which a system was analyzed. Thus, the complete list of protection sets provides the security analyst with a set of options that can be used to develop an optimal protection strategy. Note that while this analysis technique has traditionally been applied to physical security, there is no reason that the locations have to be physical. In a computer network environment, one could also use virtual locations to accomplish similar results.

The cuts sets produced by a fault tree analysis (as derived from the common object model) can be used in a variety of other valuable ways. When these cut sets are quantified, they produce an estimate of the likelihood that the system will be placed into the undesired state for which it is being analyzed. Since likelihoods of individual failure events within the cut sets are varied, the list of cut sets can be requantified to determine new estimates of the likelihood that the system will be placed in this undesired state. Since this likelihood is often important for either risk or reliability analyses, and since the goal of such analyses is to identify which of a system's components or procedures should be modified in order to produce the greatest degree of system improvement, the quantification of cuts sets can be used in the scoring ("utility") function for an optimality algorithm. Genetic optimization[8] is a particularly effective method for assessing a large number of proposed changes and improvements to a system in order to determine which combination of changes will be most cost-effective. In addition, if one considers the mean time to repair a particular component, similar techniques can be used to determine the optimal spare parts inventory and maintenance strategy for the system given a variety of cost constraints. Thus, while these optimization results cannot be directly derived from the common object model, the logic models derived from that common object model are critical for the development of a reliability-based utility function for these powerful optimization techniques.

Finally, a key component in most modern surety analyses is the assessment of uncertainty. This project did not develop a unified method for performing uncertainty analyses directly on the common object model. However, the methods for performing Monte Carlo-based uncertainty analyses for both event trees and fault trees are well known and have been embodied in separate software. Sandia's Latin hypercube sampling software (LHS[9]), fault tree analysis software (SABLE[10] and TEMAC[11]), and event tree analysis software (SETAC[12]) all embody this technology. In addition, it would not be difficult to place the common object model within the uncertainty analysis framework that is driven by the LHS software. Therefore, while uncertainty analysis was not a specific focus of this project, the methods described here can be used to predict the uncertainty in the surety results that are derived from the common object model.

5.5 Summary

This section has described how a variety of common surety evaluation models can be derived from the common object model described in previous chapters. This project demonstrated that the common object model directly supports the development of inductive surety models, including failure modes and effects analysis, hazards and

operability analysis, and event tree analysis. We also demonstrated that the common object model can be interrogated to produce the most common deductive surety model, the fault tree analysis, and, with relatively simple extensions, should also support influence diagrams. A variety of other techniques (which are not directly supported by the common object model) can be exercised based on results that are derived from the common object model. These include vital area analysis, genetic optimization, and uncertainty analysis. Finally, since the causality and behavior of the system are embodied in the common object model, it is relatively trivial to exercise the common object model as a discrete-event simulator. The results of such a simulator are very important to surety analysts as they seek to explore the behavior of a system relative to specific threats and stimuli.

From this discussion and the variety of methods supported by the common object model, one can see the power of this object-based model for surety analysis. Without using a common object model, one is required to construct and validate every instance of each of these different types of models by hand – obviously a time-consuming and potentially expensive process. However, by making use of a common object model that is created once and interrogated many times, one can quickly generate many logic models without going through a hand construction phase. This enables the surety analyst to either perform current generation analyses much more quickly, or to generate a wider variety of surety models for analysis under the current time constraints. This would enable the analyst to develop a better understanding of a system, its reaction to stimuli, and its vulnerabilities than would have been possible using previous methods. Clearly, both of these are worthwhile objectives and valuable results from this project.

5.6 References

1. McCormick, N.J., *Reliability and Risk Analysis: Methods and Nuclear Power Applications*, Academic Press, New York, 1981.

2. Greenberg, H.R., and Cramer, J.J., editors, *Risk Assessment and Risk Management for the Chemical Process Industry*, Van Nostrand Reinhold, New York, 1991.

3. Cramond, W.R., *et al.*, *Probabilistic Risk Assessment Course Documentation*, SAND85-1495, NUREG/CR-4350, 7 volumes, Prepared by Sandia National Laboratories for the U.S. Nuclear Regulatory Commission, Washington, DC, 1985.

4. Roberts, N.H., Vesely, W.E., Haasl, D.F., and Goldberg, F.F., Fault Tree Handbook, NUREG-0492, U.S. Nuclear Regulatory Commission, Washington, DC, 1981.

5. Jae, M., and Apostolakis, G.E., "The Use of Influence Diagrams for Evaluating Severe Accident Management Strategies," *Nuclear Technology* **99**:142-157, August 1992.

6. Shlaer, S., and Mellor, S.J., *Object Lifecycles: Modeling the World in States*, Prentice Hall, Englewood Cliffs, NJ, 1992.

7. Stack, D.W., and Hill, M.S., A SETS User's Manual for Vital Area Analysis, SAND83-0074, NUREG/CR-3134. Prepared by Sandia National Laboratories for the U.S. Nuclear Regulatory Commission, Washington, DC, 1984.

8. Painton, L., and Campbell, J., "Genetic Algorithms in the Optimization of System Reliability," *IEEE Transactions on Reliability*, Special Issue on Design, **44**(2), 172-178, 1995.

9. Wyss, G.D., and Jorgensen, K.H., A User's Guide to LHS: Sandia's Latin Hypercube Sampling Software, SAND98-0210, Sandia National Laboratories, Albuquerque, NM, February 1998.

10. Hays, K.M., Wyss, G.D., and Daniel, S.L., A User's Guide to SABLE 2.0: The Sandia Automated Boolean Logic Evaluation Software, SAND96-0842, Sandia National Laboratories, Albuquerque, NM, 1996.

11. Iman, R.L., and Shortencarier, M.J., A User's Guide for the Top Event Matrix Analysis Code (TEMAC), SAND86-0960, NUREG/CR-4598. Prepared by Sandia National Laboratories for the U.S. Nuclear Regulatory Commission, Washington, DC, 1986.

12. Wyss, G.D., Daniel, S.L., Hays, K.M., and Brown, T.D., Application of the ARRAMIS Risk and Reliability Software to Nuclear Accident Progression, Presented at the 1997 winter meetings of the American Nuclear Society, November 16-20, 1997.

6 Example Problem

6.1 Construction of a Functional Common Object Model

Let us now apply the object modeling methodology described earlier to a simple electromechanical system. Consider a system that would move water from a reservoir to some destination where it would be put to beneficial use. Ideally, this system would only provide water when instructed to do so by a human operator. The laws of physics being what they are, the system would be required to make use of some power source in order to move the water. In addition, the system might, in a fault condition, move water from the reservoir to some location other than its intended destination. This condition we will call "spillage." Figure 6-1 is a system structure diagram for a system that would meet these functional specifications. The alert reader will notice that this diagram contains information about abnormal behaviors, while the modeling methodology described in Chapter 4 showed the analyst completing the normal behavior model before examining abnormal behaviors. Both analytical methods are valid. The choice of incorporating these behaviors initially or in the later specific search for abnormal behavior is largely a matter of the analyst's personal preference.

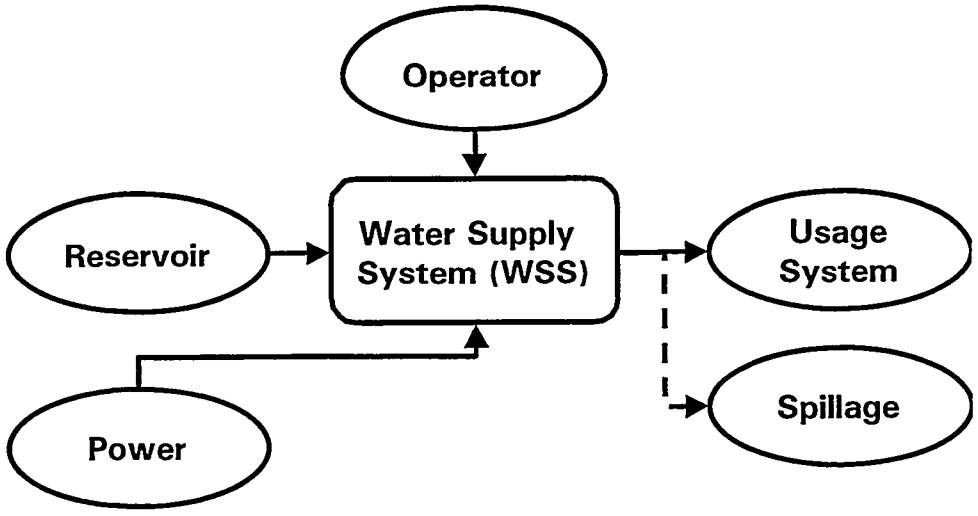


Figure 6-1. Structure Diagram of Water Supply System

At this level of abstraction (i.e., a high-level purely functional model), the system consists of only one functional box – the water supply system itself. The other entities in the diagram embody those external objects with which this system will be required to interact. These represent the boundary conditions under which the water supply system will be required to operate. The dynamics of this system as it interacts with its environment can be embedded in an interaction diagram, as shown in Figure 6-2. In this

figure we see the types of flows that move between the various objects inside and outside the system. While the actual discrete values that will be used to represent these flows are not shown in the interaction diagram, they are explicitly listed in Table 6-1. In some object-oriented analysis methodologies, this table would be called the object communication model. Note that this table contains two "random failure" events. These embody mechanisms by which abnormal behavior can manifest itself within the system.

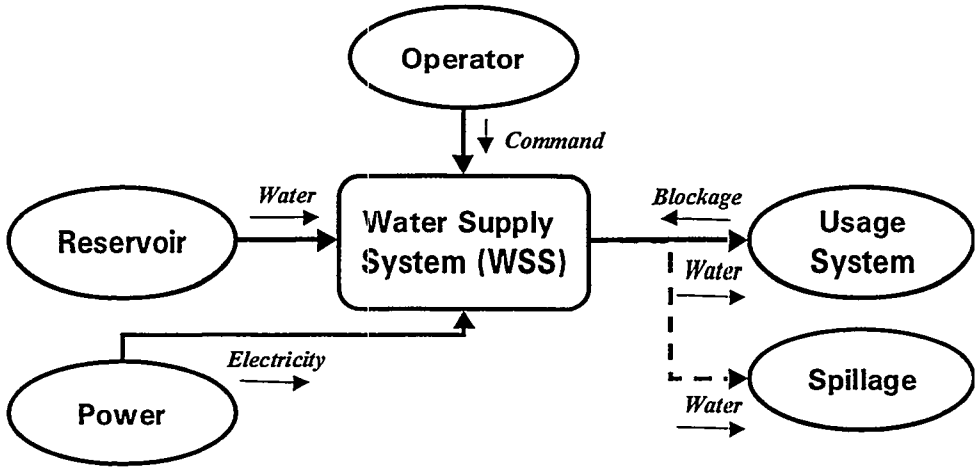


Figure 6-2. Interaction Diagram for the Water Supply System

The next step in constructing an object model for the water supply system is to develop a state transition diagram for the system. Since this is a purely functional representation of the system (granted it does contain a few high-level abnormal behaviors), the state transition diagram will reflect this high-level abstraction. Since no physical instantiation of the system has yet been specified, all of the states and transitions in this diagram will represent descriptions of different functional configurations for the water supply system. The state transition diagram will embody all of the events described in the interaction diagram – even those "random" events that do not have an identified source within the interaction diagram. One appropriate state transition diagram for the system is shown in Figure 6-3. Note that, as with many logical modeling methodologies, it is sometimes possible to describe the same system behaviors using more than one logical model. This means that the various diagrams generated for this example problem are not necessarily unique.

Each of the four states in this state transition diagram represents a particular body of functionality that is active only when the system exists in that state. In other words, the system responds to both event stimuli and the values of particular attributes in different ways, depending on the system's state when that input is received. The analyst must now construct a data flow diagram (sometimes also referred to as a process flow diagram) to describe the system's behavior in each state. In addition, for every transformation found in each data flow diagram, the analyst must specify a truth table that describes how that

entity transforms its input values into output values. The combination of transformations, truth tables, and state changes is unique for each state in the state transition diagram, and embodies the behavior of the system when it is in that state.

Table 6-1. Message Values in the Water Supply System

From/To	Type of Flow	Name	Characteristics (messages, events)
Operator/WSS	Command	ON	Binary, event
		OFF	Binary, event
Reservoir/WSS	Water	Available	Binary, message
		Unavailable	Binary, message
WSS/Usage Sys.	Water	Flow 0	Multivalued, message
		Flow Low	Multivalued, message
		Flow adequate	Multivalued, message
Power/WSS	Electricity	Inadequate	Multivalued, message
		Inappropriate	Multivalued, message
		Available	Multivalued, message
Usage Sys./WSS	Blockage	Accepting	Multivalued, message
		Rejecting	Multivalued, message
		Restricting	Multivalued, message
WSS/Spillage	Water	Spillage 0	Multivalued, message
		Spillage positive	Multivalued, message
(Random)/WSS	Random failure	Spill	Event – Randomly generated
(Random/Internal)/ WSS	Random or internal failure event	Broken	Event – Randomly or internally generated

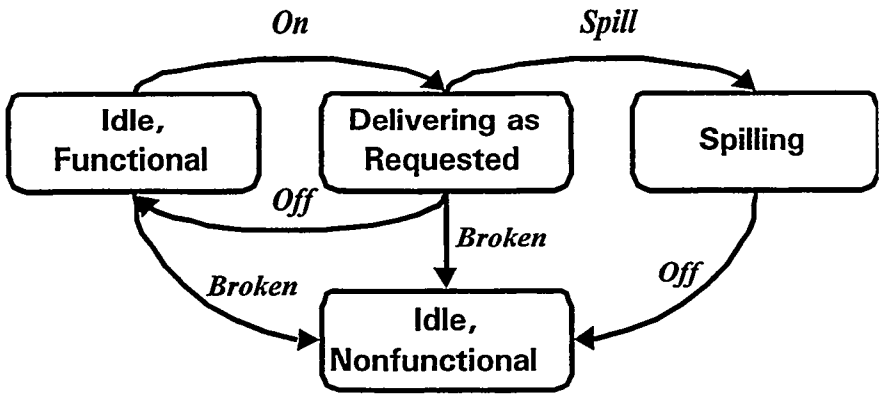


Figure 6-3. State Transition Diagram for the Water Supply System.

The water supply system engages in trivial responses in two of the four states. In both the "idle, functional" and the "idle, nonfunctional" states, the system simply receives the event that causes the state transition and responds by setting the flow of water to the usage system to zero. The system's behavior in the "spilling" state is only slightly more complicated: the system receives the "spill" event and responds by setting the flow of water to the usage system to "flow low" and the flow of water to the spillage object to "spillage positive." This represents a view of the system in which a nontrivial leak will still provide water to the usage system – albeit a reduced amount. Should such spillage occur, we assume that it would not prevent the operator from manually turning off the system, causing its transition to the "idle, nonfunctional" state. Such assumptions are likely to be appropriate early in the design phase in the system. They must be revisited later once a particular physical instantiation of the system is selected. The data flow diagrams and truth tables for these three states are shown in Figure 6-4. In these truth tables, the *"True"* designator simply implies that the transformation on this line of the truth table applies every time this transformation is exercised regardless of the inputs with which it is presented.

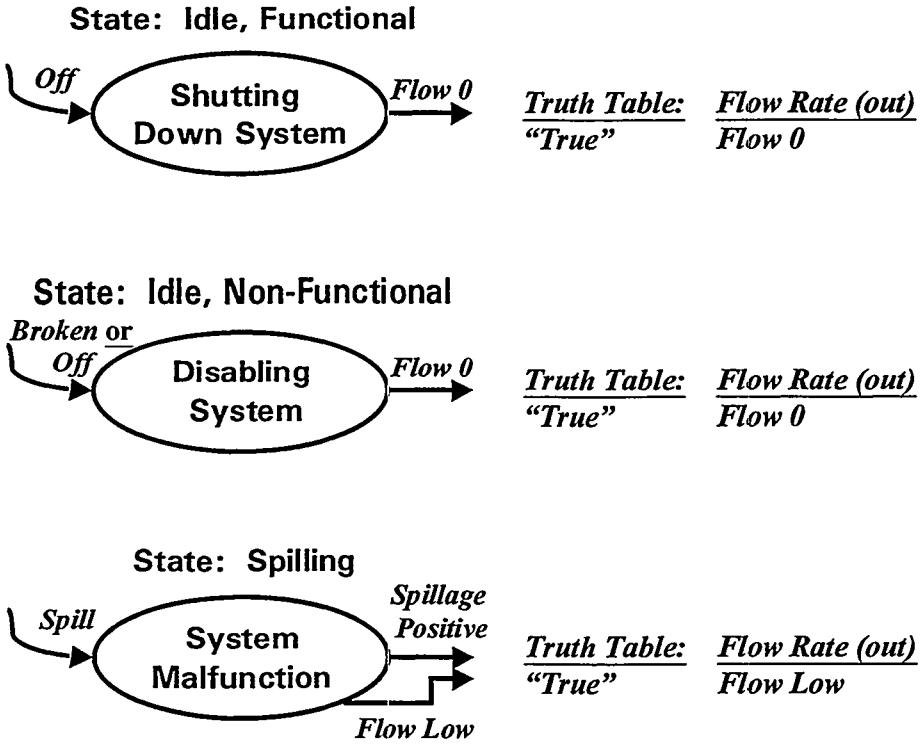


Figure 6-4. Data Flow Diagrams and Truth Tables for the Three Simplest States.

The fourth state, "delivering as requested," represents a more complex body of behavior. Within this state, one must consider not only whether water and appropriate power are available to the system, but also whether water is being accepted by the usage system.

The output from this state is the discrete attribute value for the flow rate of the water that is delivered to the usage system. It is also assumed at this early design phase that the application of "inappropriate" power to the system will in some way damage it so that it is unable to supply any water to the usage system (e.g., inappropriate electrical power may damage the electric motor for a pumping system). It is possible to embody the behavior envisioned for this system state in a number of different ways. It could, for example, be embodied in a single transformation that would receive inputs from the power, reservoir, and usage system external objects and in a single truth table would determine the flow rate out of the system and whether the "broken" event would be generated. However, the logic of such a truth table would be more complex than necessary. A second way to embody this behavior would be to create two transformations: one that determines the maximum flow that the system can potentially produce ("potential flow"), and a second that determines the actual flow produced by the system after consideration of any possible blockage found in the usage system. We have chosen to represent the system behavior for this state using this dual-transformation approach both because it is simpler to understand and because it will enable us to demonstrate more clearly how one applies deductive logic to such a multipart data flow diagram later in this chapter. Figure 6-5 shows the data flow diagram that embodies this behavior. The truth tables that support this diagram are found in Table 6-2 and Table 6-3.

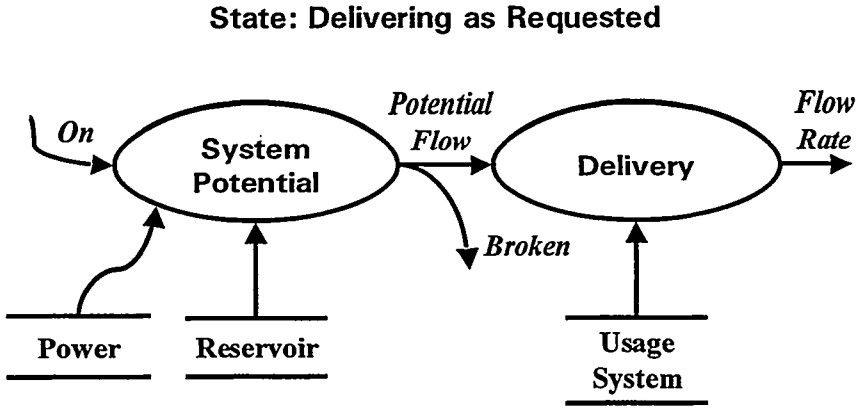


Figure 6-5. Data Flow Diagram for the "Delivering as Requested" State

Note that each of the truth tables in Table 6-2 and Table 6-3 takes into account all possible combinations of the input flow attributes (even though some were handled through a catchall "don't care" designation). This should be characteristic of all well-constructed transformation truth tables. Note also that these truth tables are completely deterministic. There are no instances where an output designator can go to more than one possible value based on a stochastic behavior found in some portion of the system. While not seen in this example, such stochastic behavior can be embedded in the system and utilized in the surety logic models that are derived from the object model.

Table 6-2. Truth Table for the "System Potential" Transformation

Power (input)	Reservoir (input)	Potential Flow (output)	Event (output)
Appropriate	Available	Full	–
Inadequate	X	0	–
Inappropriate	X	0	Broken
X	Not Available	0	–

Note: X = "don't care" (any and all values of this input satisfy the condition).

Table 6-3. Truth Table for the "Delivery" Transformation

Potential Flow (input)	Usage System "Blockage" (input)	Flow Rate (output)
Full	Accepting	Flow adequate
Full	Restricting	Flow low
Full	Rejecting	Flow 0
0	X	Flow 0

Note: X = "don't care" (any and all values of this input satisfy the condition).

We have now completed the development of the common object model for the high-level description of the water supply system. The model is complete because each state in the state transition diagram is now populated with a data flow diagram, and each transformation in each data flow diagram is now populated with a truth table.

6.2 Extraction of Fault Tree Models

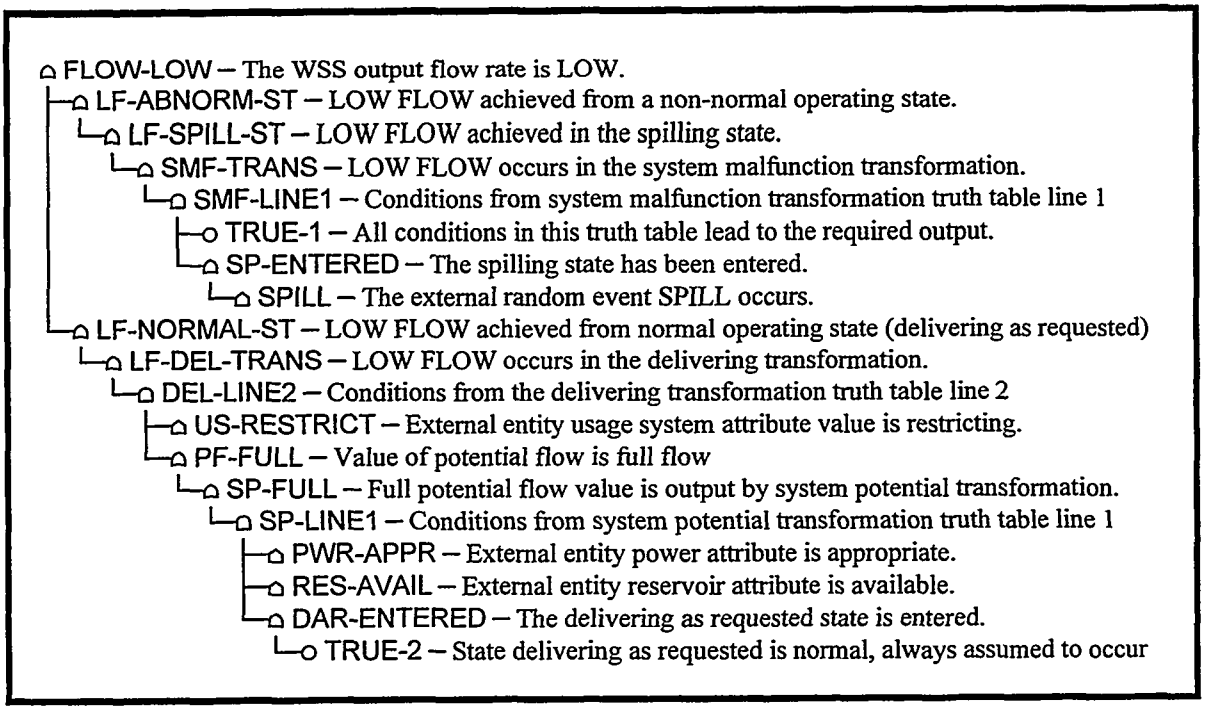
Let us now extract a fault tree surety model from the common object model described in Section 6-1. First, let us extract a fault tree to describe the condition in which the system produces a flow rate that is nonzero but also inadequate for the needs of the usage system. This corresponds to the single value of "flow low" for the flow rate attribute. We will construct this fault tree in accordance with the methods and rules in Appendix A.

The first step in fault tree development is to select a "normal" state for the system. We will select "delivering as requested" as the normal state, with normal attribute values being as follows: water from the reservoir is available, the flow rate of water to the usage system is adequate, electricity is available, the usage system is accepting water, and the spillage of water is zero. For this system, there are no states that are either impossible or out of bounds as initial conditions.

The next step is to define the objective of the fault tree analysis. In this case, our objective is to develop a fault tree that represents the ways that the system can produce a

flow rate that is nonzero but is also inadequate to supply the needs of the usage system. This corresponds to the single discrete attribute value of "flow low" for the output of the water supply system. This single attribute value will become the top event in the fault tree. Since it is a single event, it is not necessary to further process this description to obtain "objective cuts sets." According to our rules, the top event of the fault tree is always an OR gate. This OR gate represents the first entry in the fault tree. The completed fault tree is shown in Table 6-4. The reader will find it useful to refer to this table as the construction of the fault tree is described in the following paragraphs.

Table 6-4. Fault Tree for the "Flow Low" Condition



Given the top event of the fault tree, we must now interrogate the object model to determine all of the behaviors and events that contribute to this top event. According to the rules, this top event OR gate is expanded to include two inputs: one that represents the achievement of the "flow low" condition from the normal operating state, and a second that represents the achievement of that condition from any non-normal operating state. Each of these inputs is embodied in the fault tree as an OR gate. Let us first consider the non-normal operating states.

In this object model, the flow rate to the usage system manifests itself as a result from the data flow diagrams for each of the system's states. The truth tables for both of the system's idle states contain only the result "flow 0," but the truth table for the spilling system state does contain the "flow low" result. Under the general rules, we add an OR gate to indicate that the condition we are seeking can be produced in the "spilling" state,

which could, in theory, produce this attribute as the result of more than one transformation. In this case, only the "system malfunction" transformation causes this condition. It is represented by an OR gate in the fault tree because more than one logic line in that transformation's truth table might cause this outcome. Now each line in the truth table that generates this result will be represented as an input to this OR gate. In this case, there is only one such line, so the OR gate has only one input. While an OR gate with the single input may seem wasteful and counterintuitive, it is retained in the analysis as an aid in understanding the logic process used to generate the fault tree. We now embody the logic from this line in the truth table as an AND gate. Since the logic in this line of the truth table is simply "true," the output we are seeking is generated every time the system enters this state. Therefore the inputs to the AND gate are simply an "always true" basic event (as a placeholder to represent the line of the truth table), and a gate indicating that an event has occurred to cause the system to enter this state. This gate will be an OR gate because it is possible, in a general situation, for several events to cause this state transition, and each such event will cause the state transition by itself. In this situation, the only event that can cause the state transition is the "spill" event. This event is only randomly generated, and is not generated as a result of some other behavior either inside or outside of the system. Thus it is represented as a primary event that completes this section of the fault tree.

We apply similar rules to the normal operating state for the system. Here the flow rate to the usage system manifests itself only as an output from the delivery transformation. We place an OR gate in the fault tree to represent the fact that it is found in the truth table for this transformation. Each line in the truth table that results in a "flow low" output will be an input to this OR gate. In this case, there is only one such line. Again, it is represented by an AND gate with inputs representing full potential flow and a restricting condition in the usage system. Since the restricting condition in the usage system is generated by an entity that is outside of the system, this condition is represented as a primary event in the fault tree and is not expanded further. Potential flow, however, is itself the result of a separate transformation ("system potential"). This transformation is therefore represented as an OR gate in the system because it represents the result of a truth table. Only one line in the system potential truth table produces the result of full potential flow. The conditions from this line are embodied in an AND gate that forms the sole input to this truth table's OR gate. These conditions are: the system enters the "delivering as requested" state, power is appropriate, and the reservoir is available. The power and reservoir conditions are generated by entities that are outside of the system, so they are represented as primary events in the fault tree. Entry into the "delivering as requested" system state is represented as an OR gate because it could, in theory, be caused by several different events. However, since this state is the normal system state, we do not model how the system might achieve this state – we simply assume that it occurs *because* it is the normal state. Therefore, the input to this OR gate is an "always true" basic event, which serves as a placeholder to note that this is part of the normal state.

We have now completed the fault tree because there are no longer any unresolved inputs to any AND or OR gates in the model. This fault tree could be fed to traditional fault tree analysis software for processing and analysis. While this is a trivial example, it provides

a simple environment for demonstrating the rules of fault tree generation found in Appendix A.

Let us now continue this example by building a larger fault tree. We will not describe the construction of this fault tree in as much detail as the previous example, but rather will highlight important points in the development. Once again, we will follow the rules for fault tree development as stated in Appendix A. Assume that the system has the same normal state as specified in the previous example, and that all states and conditions defined for that analysis remain valid. With this being true, our first task is to develop a definition of the objective for the fault tree. In this analysis, we will look for all of the ways that the flow rate can be less than "adequate." In other words, we will look for situations where the output flow of water to the usage system is either "flow low" or "flow zero." This verbal logical expression can be directly transformed into the top event of the fault tree as an OR gate with two inputs: flow low and flow zero. In order to simplify the description of this problem and to take advantage of the previous example analysis, we note that the flow low criterion is in every way identical to the analysis that developed the fault tree described previously (Table 6-4). While an automated fault tree generation tool would explicitly construct this portion of the fault tree, the steps required for its construction are identical to those described previously. Therefore we will represent this portion of the fault tree only using a "developed event" symbol. This indicates to the reader that the flow low portion of the tree is developed elsewhere. In most fault tree analysis software packages, having the flow low portion of the fault tree from Table 6-4 present while the main fault tree is being solved will cause the software to automatically append this small fault tree into the larger fault tree at the appropriate point.

Let us now proceed with the development of a fault tree to represent the condition of zero water flow to the usage system. As in the previous example, we first subdivide the problem between the normal and non-normal system states. This is accomplished using an OR gate. The portion of the fault tree that assesses the non-normal system states can be found in Table 6–5, while the portion that assesses the normal system state can be found in Table 6–6. We note from the system diagram that there are two non-normal states that can result in zero flow: the idle, functional state and the idle, nonfunctional state. The development for the idle, functional state closely parallels that of the spilling states from Table 6-4. The development for the idle, nonfunctional state initially follows a very similar course. We note that zero flow occurs in line 1 of the truth table for the "disabling system" transformation. That line of the truth table is satisfied for all conditions within the idle, nonfunctional state. Thus, the inputs to the AND gate representing that line of the truth table are an "always true" condition (related to conditions within that state) and an OR gate to represent all of the different ways that the idle, nonfunctional state can be entered. The inputs to this OR gate represent the point of divergence between this state and the other states we have previously analyzed.

Table 6–5. Fault Tree for the Zero Flow Condition, Part 1

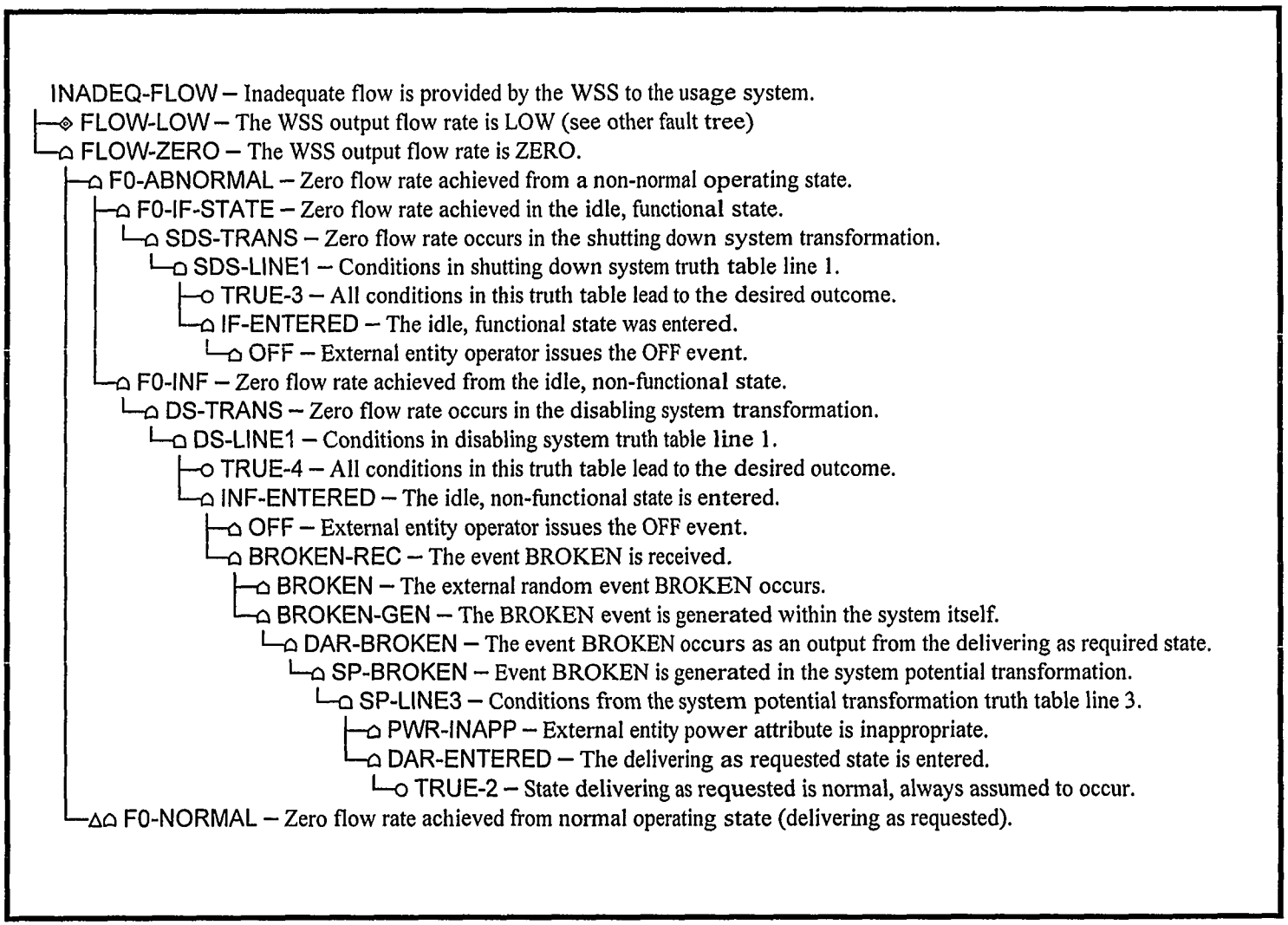
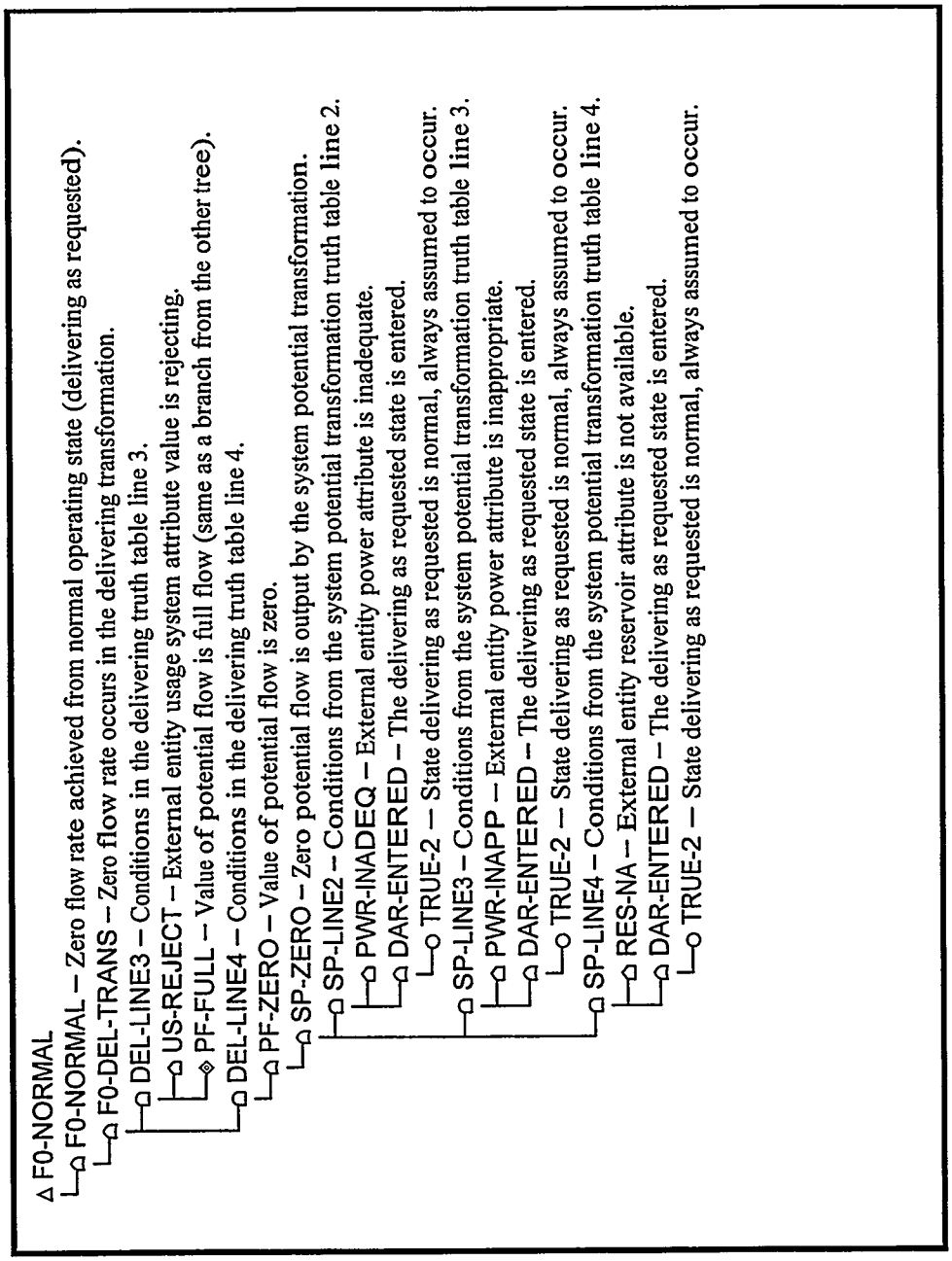


Table 6–6. Fault Tree for the Zero Flow Condition, Part 2



In previous fault tree segments, there has generally been only one way for an event that causes a state transition to be generated: some external or random entity acting on the system. In this case, however, there are two different ways that the "broken" event can occur. It can occur either as a random event or as an event that is generated within the system itself. Note that the "broken" event can be one of the outputs from the "delivering as requested" state. Therefore there must be two inputs to the OR gate that represents receipt of the "broken" event. The random "broken" event is represented as a primary event in the fault tree. The receipt of an internally generated "broken" event is represented as an OR gate, the inputs of which will be all states within the object model that are capable of producing this event. In this case, only the "delivering as requested"

state is capable of producing such an event. It is generally possible for an event to be generated by more than one transformation within a state, so the fact that this state can generate this event is represented as an OR gate with one input for each transformation that is capable of producing the event. In this example, only one transformation is capable of producing this event: the "system potential" transformation. Now, as before, the fact that this transformation can produce the "broken" event is represented in the fault tree as an OR gate because it is generally possible for more than one logical line in a transformation's truth table to cause this event. In reality, only line 3 of the system potential truth table is capable of producing this event, so only one AND gate will be required for this transformation. We can now trace back through this transformation as before to develop inputs for the AND gate. This is done in the same manner as demonstrated in previous parts of these examples. We would only note that since the "delivering as requested" state is the normal state of the system, it is always assumed to occur. Thus, the event that would cause transition to this state is represented by a simple "always true" condition.

Let us now move on to develop the fault tree that represents the normal operating state of this system: the "delivering as requested" state. We have noted that zero flow to the usage system is a possible output in this state. Since it is possible for more than one transformation to produce this condition (although it does not occur in this case), we represent zero flow from this state using an OR gate. The "delivering" transformation, which can produce the zero flow rate condition, is represented by an OR gate because more than one logic line in its truth table can (and, in fact, does) lead to this zero flow condition. Each of these logical lines is represented by an AND gate. Truth table line 3 represents a situation where the usage system is rejecting flow even though full potential flow is available. Since the details of the full potential flow condition were expanded in the previous fault tree (shown in Table 6-4), this condition is represented by a developed event in this fault tree. The reader should refer to the "PF-FULL" events in that table to review that fault tree segment.

The other truth table line in the delivering transformation that results in zero flow to the usage system is line 4, which indicates that flow to the usage system is unavailable if the value of zero is achieved for the potential flow attribute. This attribute is produced only by the system potential transformation. Both the zero potential flow attribute and the system potential transformation are represented as OR gates in the fault tree. The system potential OR gate's inputs are those lines in the system potential transformation truth table that can result in zero potential flow: line 2, line 3, and line 4. Each of these is represented by an AND gate with the appropriate attributes from external entities as their inputs. Here, as previously, entry to the "delivering as requested" state is represented by an "always true" condition because it is the normal state of the system.

6.3 Extraction of Event Tree Models

We now turn our attention to the extraction of inductive logic models such as event trees from the common object model described at the beginning of this chapter. For this example, we will extract two different event tree models: one that models only the effects

of random events on the system, and a second that models the effects of both random and external deterministic events on the system. We will construct these event trees in accordance with the methods and rules in Appendix A.

The first step in the development of an event tree for a system model is to specify the initial state of that system model. This includes both the states of the various model objects as well as values for all discrete attributes. As in the first fault tree example, we will select the "delivering as requested" state. The normal attribute values once again will be as follows: water from the reservoir is available, electricity is available, the usage system is accepting water, and the spillage of water is zero. Under these conditions, the common object model dictates that the water supply system is delivering adequate flow to the usage system. Again, for this system, there are no invalid states or conditions.

The next step in the development of an event tree is to select an initiating event for the event tree. For this analysis, we will assume an initiating event of the system operating in a normal steady state, which occurs immediately after the "ON" event is received by the system from the operator.

The methodology described in Appendix A dictates that the next step in the event tree construction process is to select a starting point for the event tree model. Since we are starting from a steady state, we must determine which object is to be used for the starting point of the event tree. For this simple model, the answer is obvious because the water supply system is the only object in the model.

We now examine the object model to determine which events are to be included in the event tree. We are considering only random events that can affect the normal "delivering as requested" state. Let us consider the messages that are available within this system as found in Table 6-1. Here we see two randomly generated events: "spill" and "broken." According to the state transition diagram shown in Figure 6-3, either one of these events will cause the system to transition out of the desired state. In other words, the system will respond to both of these events in the "delivering as requested" state (neither of these events is "ignored" by the system when it is in this state)(over) for addition at this point). Therefore, these two events will form the branch points for the event tree, and will be listed across the top of the event tree diagram, as seen in Figure 6-6. Since each of these events is random (i.e., not generated within the system), the ordering of events in the event tree is arbitrary.

The final step for generating an event tree is to follow the effects of the specified events as they evolve into scenarios for the object model. The first branch point in the event tree is the occurrence or nonoccurrence of the "broken" event. Let us first consider the scenario where broken occurs. According to Figure 6-3, this event causes an immediate transition to the state "idle, nonfunctional," which is an absorbing state (i.e., no transitions out of that state are possible). Therefore, under the assumptiions of this simplified model, it is irrelevant whether the "spill" event occurs once the broken event has already occurred, so the event tree does not branch further on that event. The data flow diagram for the idle, nonfunctional state (found in Figure 6-4) indicates that the

system is now producing zero flow to the usage system. This situation is represented as the third path in Figure 6-6.

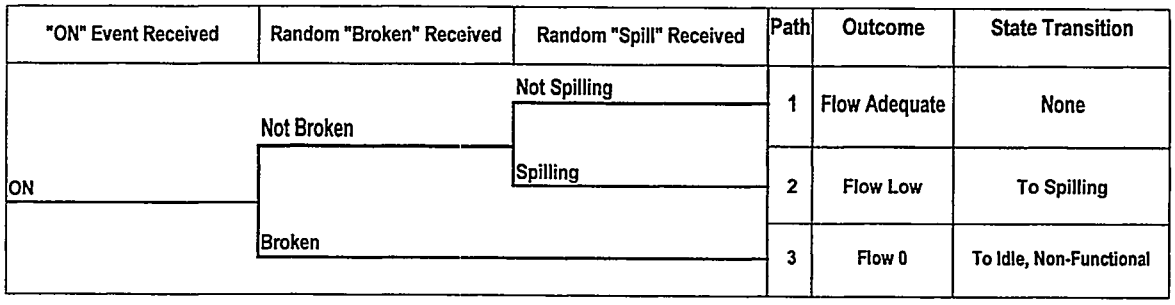


Figure 6-6. Event Tree Model Using Only Random Events

The next scenario involves the nonoccurrence of the broken event. Since nothing has yet occurred to cause the system to vary from its initial state, everything in the system is still set to its initial condition. We now consider the occurrence or nonoccurrence of the "spill" event. According to Figure 6-3, this event causes an immediate transition of the system into the "spilling" state. While spilling is not an absorbing state, there are no random events that can cause the system to transition out of that state. Thus this path is complete and leads to a system that is spilling its fluids and producing a low flow rate, as described in the data flow diagram for the spilling systems state in Figure 6-4. The second path in Figure 6-6 represents this scenario.

We must now consider the nonoccurrence of the spill event (the option that was not chosen for the second scenario above). For this scenario path, we examine the situation where neither the broken event nor the spill event occurs. Here nothing occurred to cause the system to vary from its initial state. In addition, there are no more random events to be considered in the analysis. Therefore, the system remains in its initial condition of delivering adequate flow to the usage system without spilling. This can be seen in the first path in Figure 6-6. Since there are no more random events to consider, and all path options have been exhausted for the specified random events, the construction of the event tree is complete.

Let us now consider an event tree model that examines the effects of both random and deterministic influences on the system. We identified the random influences on the system in the previous model: the broken event and the spill event. The deterministic influences on this system consist of the discrete attribute values that can be achieved for those entities that are external to the system: the reservoir, the electric power system, and the usage system.* The reservoir communicates with the water supply system through the

* Note that for this analysis we have neglected the effect of the operator since the operator communicates with the system by issuing events rather than by changing discrete attribute values. We could easily include this effect by placing the operator's events ("on" and "off") in the event tree in the same manner used for the broken and spill events. However, this would have unnecessarily complicated the example without providing significant new insight.

water attribute, which can have values of "available" and "unavailable." The electric power system communicates with the water supply system through the electricity attribute, which takes on values of "available," "inadequate," and "inappropriate." Finally, the usage system communicates with the water supply system through the "blockage" attribute, which can take on values of "accepting," "rejecting," and "restricting." These flows and their possible values are documented in Table 6-1.

We must now establish the order in which the various events in discrete attribute values will be considered in the event tree model. Since the random events cause an immediate transition of the system into a new state, they are placed first in the event tree. The remaining elements – the discrete attribute values – should be placed in the tree in the order in which they are encountered within the common object model. This can be determined by examining the data flow diagram for the delivering-as-requested state (the initial condition for this system), as shown in Figure 6-5. In that figure, the power system and the reservoir contribute to the first transformation bubble, while the usage system contributes to the second transformation. This implies that the effects of power and the reservoir should be considered before the effects of the usage system. Since each transformation within the system model is simply a truth table, where order is unimportant, the discrete attribute values that contribute to each transformation can be entered in the event tree in an arbitrary order. The ordering of events that was selected for this analysis can be seen in the event tree solution in Figure 6-7.

We are now ready to generate the actual event tree by following the effects of the specified events as they evolve into scenarios. The development of scenarios for the first two events (the "random" events) proceeds in a manner identical to that of the previous example. Again, the scenarios that result from receipt of the broken and spill events cause transitions to other states for which the selected discrete attribute values have no effect. This can be seen from the data flow diagrams in Figure 6-4. Therefore, these scenarios experience no further branching and proceed immediately to the end of the model, where their outcomes are assessed. These scenarios are represented by paths 7 and 8 in Figure 6-7.

We must now extend the path created by the nonoccurrence of both the spill and broken events. This path is extended by considering the possible values for the discrete attribute values imposed on the system by external entities. The first such entity to be considered is power. Let us first consider the value " inadequate," which becomes manifest in the system potential transformation that is documented in Table 6-2. This situation provides zero potential flow regardless of the reservoir condition, and, according to Table 6-3, results in zero flow rate to the usage system regardless of the value of blockage. Thus the path involving inadequate power experiences no further branching and results in zero flow, as seen in path 6 in Figure 6-7. Similarly, a value of "inappropriate" for power manifests itself not only in zero potential flow (regardless of reservoir conditions), but also causes the generation of the broken event. According to Figure 6-3, this cause is an immediate transition to the idle, nonfunctional state. In this state, the system does not

respond to reservoir or blockage conditions or to other events and always results in zero flow. This scenario is represented by path 5 in Figure 6-7.

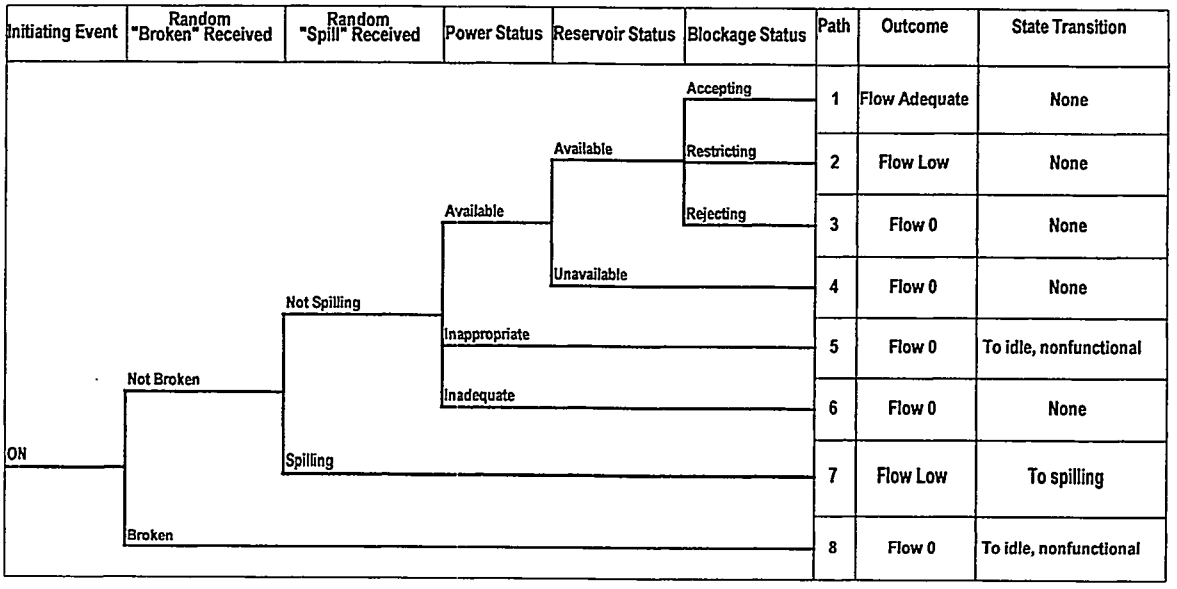


Figure 6-7. Event Tree Model Using Random and Deterministic Events

When the value "appropriate" is selected for power, Table 6-2 reveals that more than one outcome is possible, depending on the state of the reservoir. Therefore, we step forward assuming power is appropriate (or "available") to consider the various discrete attribute values associated with the reservoir. When the reservoir is "not available," potential flow is zero, and, according to Table 6-3, the only possible outcome for the flow rate delivered to the usage system is also zero. Thus, no further branching is possible based on blockage. This results in the scenario described by path 4 of Figure 6-7.

Let us now stop to review our current position in the event tree analysis: the broken and spill events have not been received, and power is available. We must now consider the other possible value for reservoir: "available." Table 6-2 tells us that appropriate power and an available reservoir lead to full potential flow. However, in Table 6-3, full potential flow can lead to three different final results, depending on the value for blockage. We will complete the event tree by considering these values. When blockage takes on the value "rejecting," the value for flow rate becomes zero. This is the end of the data flow diagram, and all random events and discrete attribute values have been considered. Therefore this path is complete. It is illustrated by path 3 in Figure 6-7. Similarly, when blockage takes on the value "restricting" or "accepting," Table 6-3 indicates the results of "flow low" and "flow adequate," respectively. These scenarios are represented by paths 2 and 1 in Figure 6-7, respectively.

A variety of other event tree models – and, indeed, many other types of inductive models – can be derived from the object model described at the beginning of this chapter.

However, owing to space and time constraints, they are not developed in this report in order that we may present the more important subjects of expanding the existing one-object model into a multiobject model, and deriving logic models from that expanded model.

6.4 Expansion of the Common Object Model

In Section 6.1, we constructed a functional, object model for a simple water supply system. That model was extremely simple in that it consisted of a single object and contained no physical components. Let us now expand that model to include a physical instantiation for this system. Consider a water supply system composed of physical components as shown in Figure 6-8. Notice that this system contains the same external entities as the original water supply system: an operator, a power supply, a reservoir, a usage system, and spillage. However, the previously posed simple functional description of the water supply system has now been replaced by physical components that include two wires, two pipes, a switch, and a pump. This simple system will enable us to demonstrate the applicability of the methods developed for this project to common object models composed of more than one object.

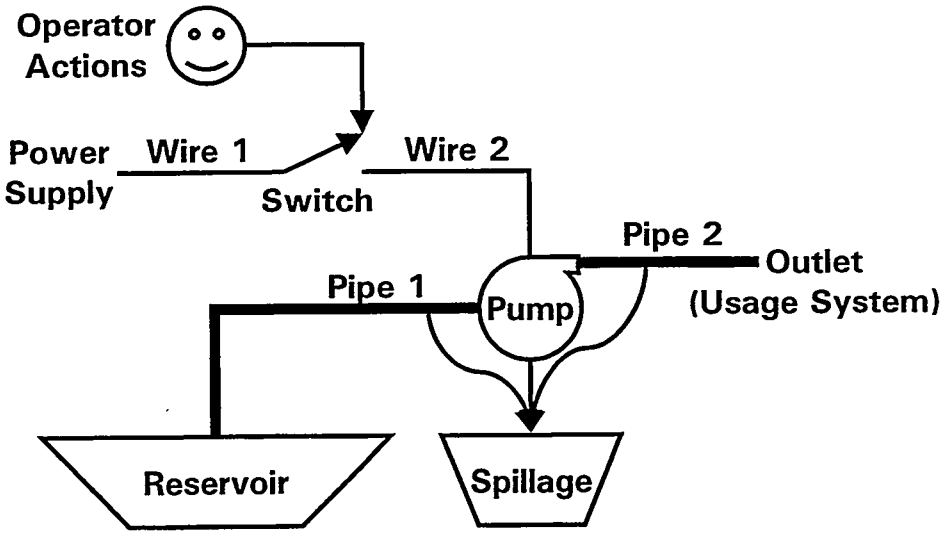


Figure 6-8. Expanded composition of the water supply system

The composition described in Figure 6-8 can be represented more formally in a system structure diagram, as shown in Figure 6-9. Here we note which objects interact with other objects and with entities from the outside world. Note that the external spillage entity is now capable of receiving flows from all system components that carry water.

This system structure diagram contains information about abnormal behaviors as did the previous simple model. These abnormal behaviors are more concrete in this case because of the physical instantiation of the system being modeled.

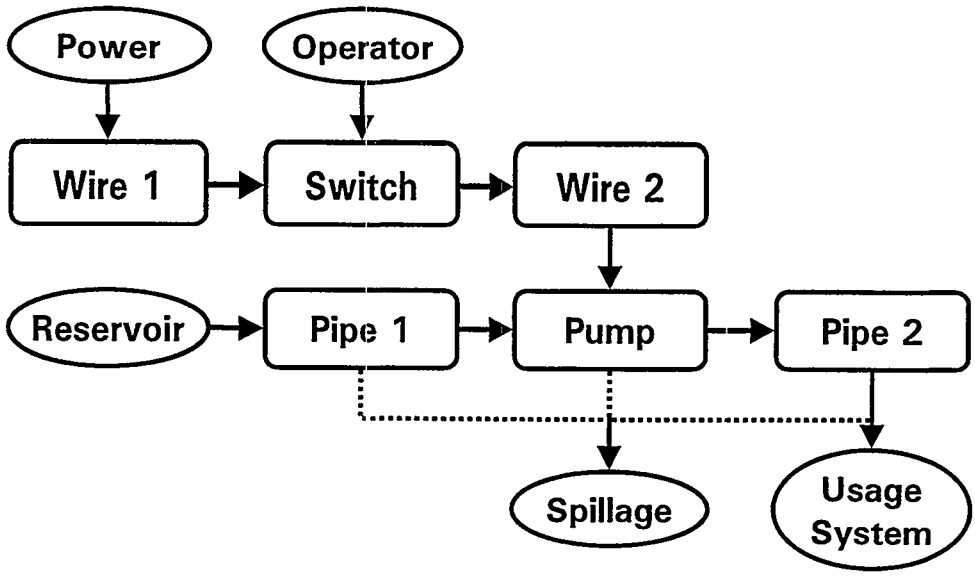


Figure 6-9. Structure Diagram of Expanded Water Supply System

The system structure diagram from Figure 6-9 can be enhanced to represent the various interactions within the expanded water supply system. These are represented by the interaction diagram found in Figure 6-10 as well as the accompanying message values for Table 6-7. This table is clearly much more detailed than the original message value table (Table 6-1). This is because there are many more interactions taking place in this expanded model than there were in the original simple model, and all interactions and their discrete attribute values must be cataloged as part of the common object model. Note also that there are many more failure events in this object model than were seen in the original simple model. These failure events are tied to the physical instantiation of individual components. For example, it is now possible for wires to experience open or short circuits, for switches to experience hardware failure, and for pumps and pipes to experience both plugging and leakage.

Specifying the interaction diagram and the message value table may seem to be an overwhelming task for even this simple system. However, this does not have to be the case. If one looks forward to the time when a system such as this might be in widespread use, it is very easy to envision libraries of generic components such as pumps, pipes, wires, switches, valves, computers, and so forth. Each of these generic components would contain its own structure diagram and message value table that could be plugged into the larger common object model once appropriate interfaces were specified. We envision a time when a systems analyst could simply drag such components onto a computer screen and connect them using graphical commands to quickly assemble an

appropriate system model. Obviously, such a library does not exist at this early point in the development of the methodology, but assembling one would add a great deal of utility to this methodology.

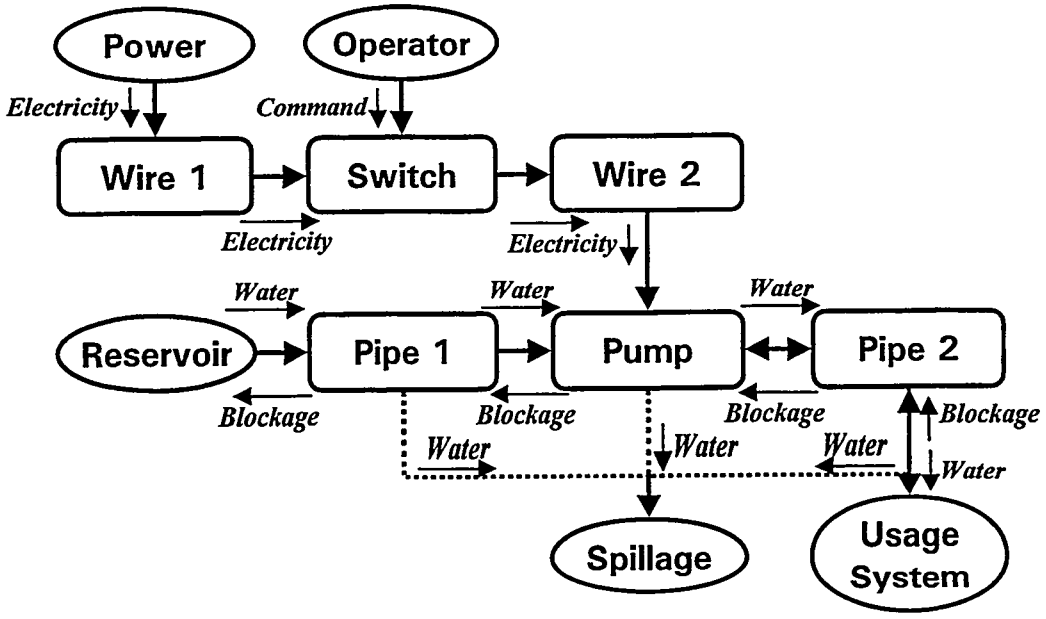


Figure 6-10. Interaction Diagram for the Expanded Water Supply System

Table 6-7. Message Values in the Expanded Water Supply System Model

From/To	Type of Flow	Name	Flow Characteristics (messages, events)
Operator/Switch	Command	On	Event, binary
		Off	Event, binary
Power/Wire 1	Electricity	Inadequate	Multivalued message
		Inappropriate	Multivalued message
		Available	Multivalued message
Wire 1/Switch	Electricity	Inadequate	Multivalued message
		Inappropriate	Multivalued message
		Available	Multivalued message
Switch/Wire 2	Electricity	Inadequate	Multivalued message
		Inappropriate	Multivalued message
		Available	Multivalued message
Wire 2/Pump	Electricity	Inadequate	Multivalued message
		Inappropriate	Multivalued message
		Available	Multivalued message

Continued next page.

From/To	Type of Flow	Name	Flow Characteristics (messages, events)
Reservoir/Pipe 1	Water	Available	Binary message
		Unavailable	Binary message
Pipe 1/Pump	Water	Available	Multivalued message
		Restricted	Multivalued message
		Unavailable	Multivalued message
Pump/Pipe 2	Water	Available	Multivalued message
		Limited	Multivalued message
		Unavailable	Multivalued message
Pipe 2/Usage System	Water	Flow 0	Multivalued message
		Flow low	Multivalued message
		Flow adequate	Multivalued message
Usage System/ Pipe 2	Blockage	Accepting	Multivalued message
		Restricting	Multivalued message
		Rejecting	Multivalued message
Pipe 1/Spillage	Water	Intact	Binary message
		Spilling	Binary message
Pump/Spillage	Water	Intact	Binary message
		Spilling	Binary message
Pipe 2/Spillage	Water	Intact	Binary message
		Spilling	Binary message
(Random)/Pump	Random or internal failure	Leakage	Event – randomly generated
		No pumping	Event – randomly generated
		Pump burnout	Event – internally generated
		Pump plugged	Event – randomly generated
(Random)/Pipe 1, Pipe 2	Random Failure	Pipe broken	Event – randomly generated
		Pipe plugged	Event – randomly generated
(Random)/Switch	Random or internal failure	Switch O, FTC	Event – randomly generated
		Switch O, FTRO	Event – randomly generated
		Switch C, FTC	Event – randomly generated
		Switch O, FTRO	Event – randomly generated
(Random)/Wire 1, Wire 2	Random Failure	Wire open	Event – randomly generated
		Wire shorted	Event – randomly generated

We will now develop state transition diagrams and data flow diagrams for the various classes of objects specified for this expanded water supply system. While it is possible to develop a wide variety of types of object models for each of these components (with varying focus and levels of detail), the object models were deliberately left simple for this example in order to keep the development of logic models as straightforward as possible. Clearly, much more detailed representations of objects such as pumps and switches are possible. The reader is invited to consider such models in the light of the logic models developed in the next section and to trace through the effects such changes would produce in those models.

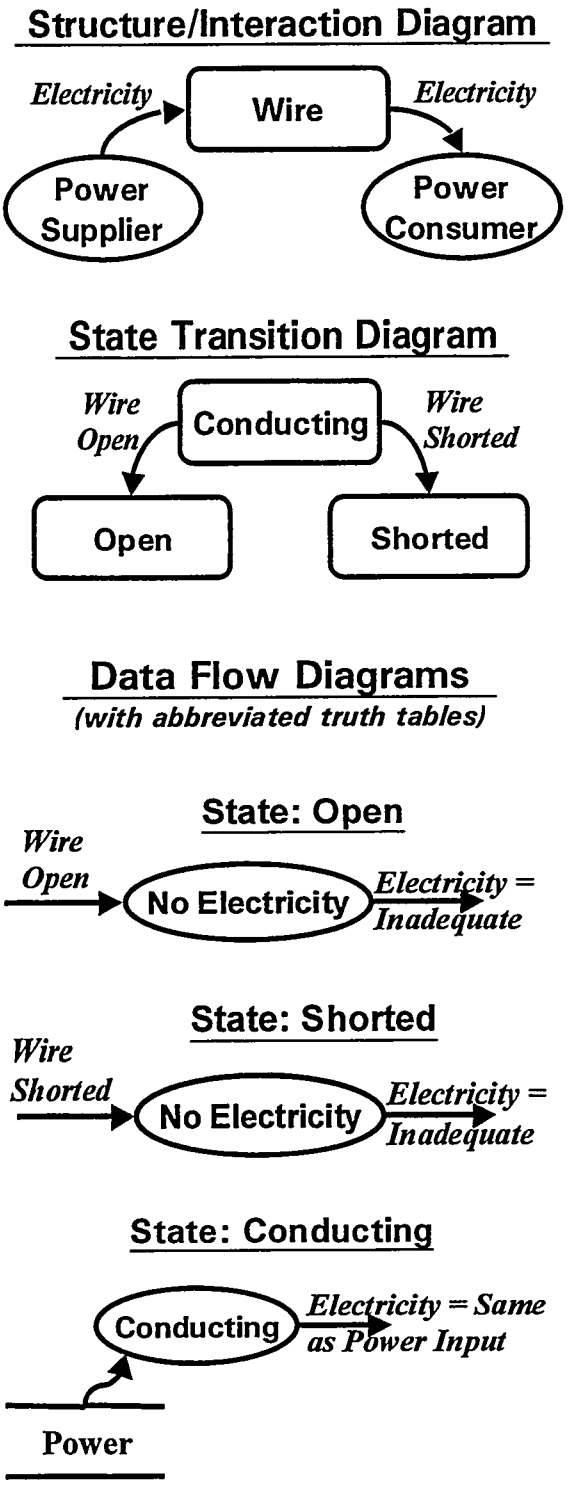


Figure 6-11. Object Model for Wires.

The first object model developed was for the generic component of a wire. This object is described by the diagrams found in Figure 6-11. Note that this object model is described using a short-hand notation in order to make it easier for the reader to visualize the entire model at once. A verbal form of the truth table has been incorporated directly on each data flow diagram because each truth table for this object is so simple.

Note that the wire can only exist in three states: conducting, open, and shorted. The events that cause transition from the normal conducting state to the other two states are random failure events. Once the wire enters into either of these states, recovery is impossible within the bounds of this model, and the wire is incapable of delivering adequate electricity to the load.

A second basic object model was developed for the generic component of a pipe. The pipe is normally capable of delivering liquids from a supplier to a consumer (in this case, that liquid is water). The pipe is also capable of transmitting blockage back from the consumer to the supplier, of being blocked itself, and of leaking (or "spilling") its contents to some generic spillage sink. This basic model is embodied in the structure and interaction diagram shown in Figure 6-12. In this model, a pipe is capable of existing in four different states: its normal "transmitting" state, plugged, restricted, or spilling. Transition from the normal transmitting state to any of the other three states occurs via a random failure event. Each of these three states is

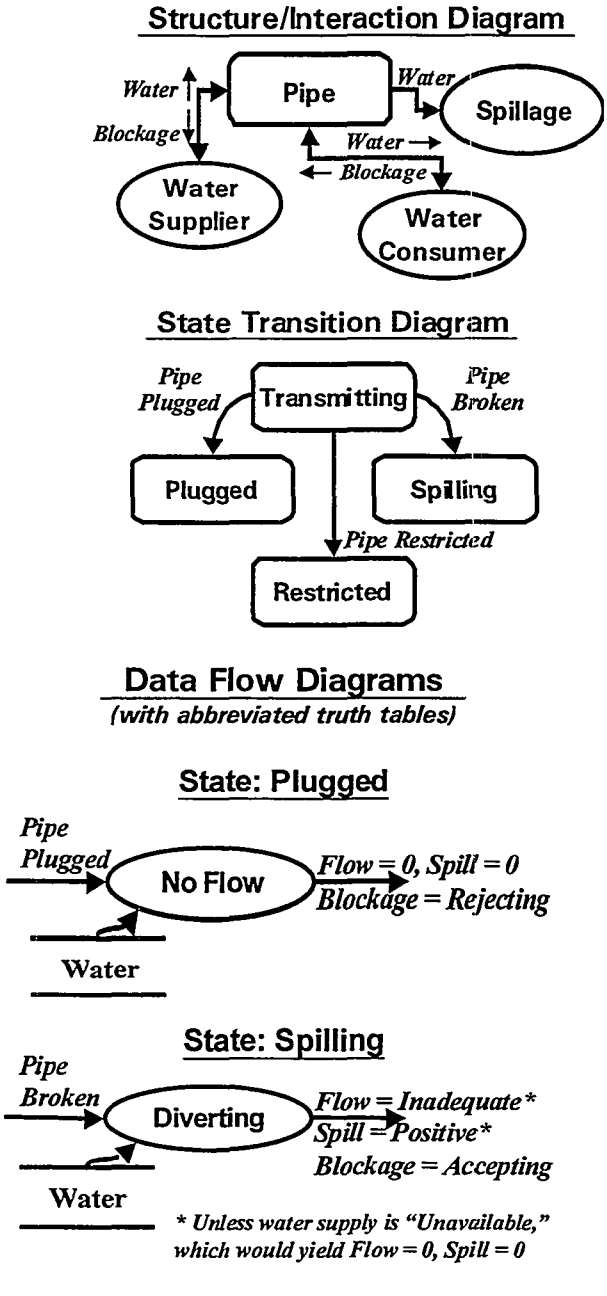


Figure 6-12. Object Model Pipes, Part 2

absorbing for the purposes of this model – it is not possible to transition back from from either plugged, spilling, or restricted into the fully transmitting state.

The data flow diagrams and the associated simplified truth tables for two of the states in the pipe object model are also shown in Figure 6-12. The behavior of a pipe in the plugged state is extremely simple: a plugged pipe cannot pass flow and rejects all liquids from suppliers. The spilling state is similarly simple: the model assumes that a significant diversion of liquid occurs so that the pipe is incapable of passing adequate flow. The pipe does, however, accept all liquids from suppliers.

The remainder of the data flow diagrams are shown in Figure 6-13. In the restricted state, the pipe is assumed to be incapable of passing sufficient flow and is acting as a restriction toward all suppliers. The more complicated behavior occurs when the pipe is in its normal transmitting state. Here the pipe is assumed to be accepting all flows from suppliers and not spilling that supply. However, depending on the availability of the liquid supply and the presence of any downstream blocking, the pipe is capable of presenting zero flow, inadequate flow or adequate flow, according to the truth table in Table 6-8.

The third generic object developed for this example problem was for a switch. This object model is shown graphically in Figure 6-14. The switch responds to the command of a human operator who issues either an "ON" or "OFF" event. Based on these commands, the switch undergoes transitions between open and closed states if it is able to do so. The generic switch object is also able to experience hardware failures. This model contains all four common types of switch hardware failures: the switch can be open but fail to remain open even in the absence of a command from the operator (O,FTRO); it can be open but fail to close

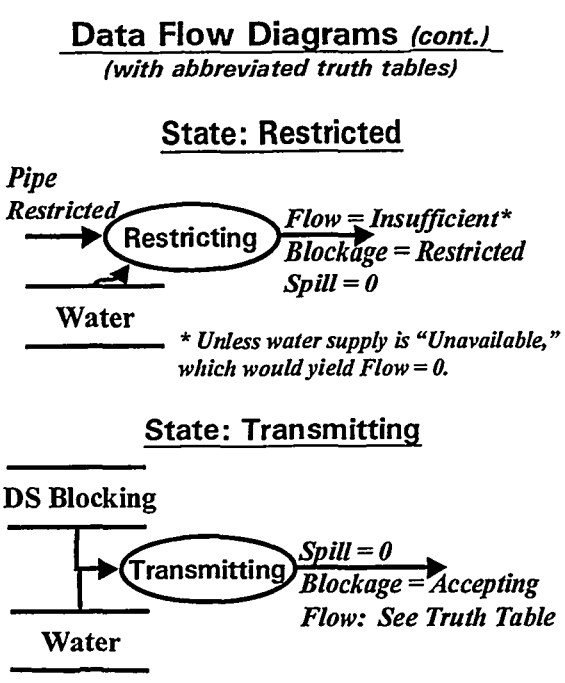


Figure 6-13. Object Model Pipes, Part 2

given an appropriate command from the operator (O,FTC); it can be closed but fail to remain closed even in the absence of a command from the operator (C,FTRC); or it can be closed but fail to open given an appropriate command from the operator (C,FTO). Each of these events is a random failure event in this object model. Note that the switch could also be moved to the wrong position – either accidentally or intentionally – by a human operator, but such an incident is properly handled in the operator object instead of the switch object.

Notice that there are two different open states in this model and two different closed states. It is assumed that the failure states (fail open and fail closed) are unrecoverable. Thus they are absorbing states in the state transition diagram. The other two states (ok open and ok closed) represent the normal operating states for the switch. The object model transitions freely between these two states based on commands from the operator.

Table 6-8. Data Flow Diagram for a Pipe in the Transmitting State.

Water Supply	Downstream Blockage	Blockage	Output Flow Rate
Unavailable	X	Same as downstream	0
X	Rejecting	Rejecting	0
Inadequate	Restricting or accepting	Same as downstream	Inadequate
Adequate	Restricting	Restricting	Inadequate
Adequate	Accepting	Accepting	Adequate

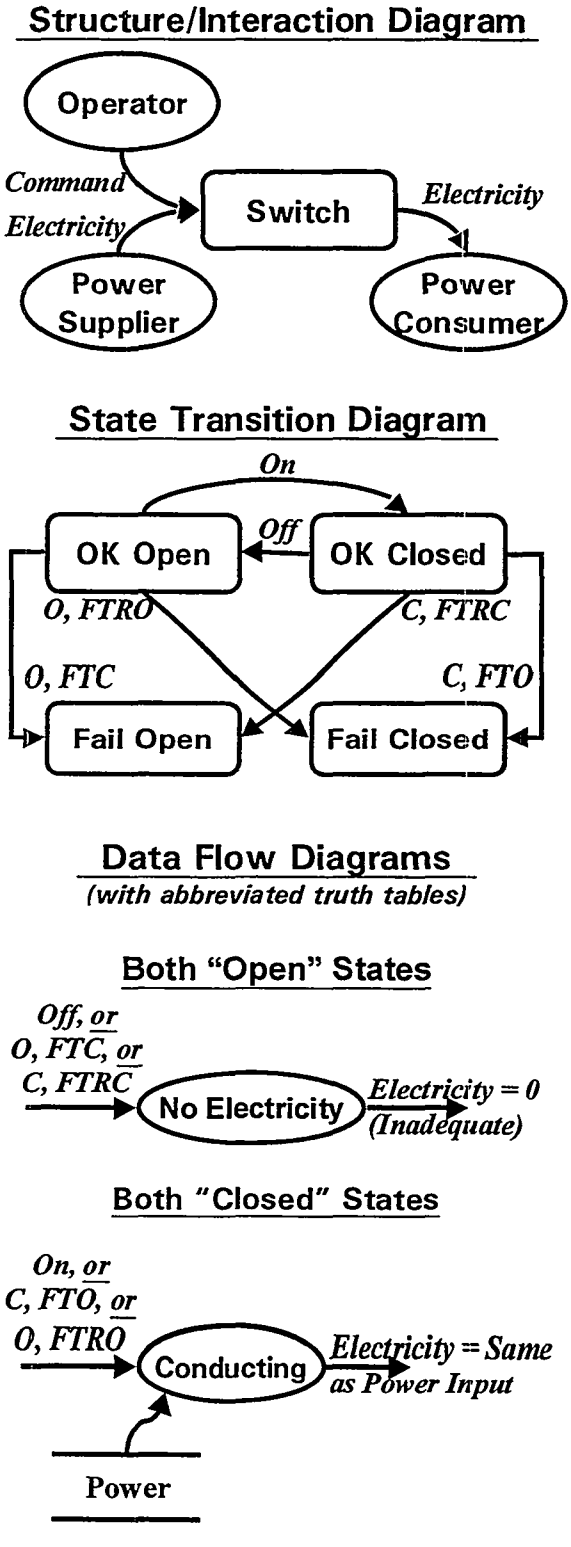


Figure 6-14. Object Model for a Switch.

Since a switch functions identically in both of the open states (it passes the electrical current from a source to a sink), both open states are represented by a single simplified data flow diagram and truth table for this model. The same is true for both of the closed states. This accounts for the compact notation used in developing the data flow diagram and truth tables for this generic object. The observant reader may notice that the switch model differs from the wire model in that the potential for a short circuit is neglected in the switch. This was done for the sake of simplicity in the switch object definition.

The fourth and final generic object model in this example problem represents a pump. The pump is a complex electromechanical device. Depending on the application, a surety analyst may wish to evaluate an object model containing a pump in a variety of levels of detail, ranging from a simple input/output model to a relatively detailed treatment, including separate models for the electrical and impeller systems. The generic object model developed here is relatively simple because it considers only three categories of flow (i.e., discrete attribute values for the flow property): zero, inadequate, or adequate. These three levels are appropriate for this simple system in order to help demonstrate the object-oriented analysis methods that are the focus of this report. The reader is encouraged to construct more complex pump object models and evaluate their impact on the overall logic models to be developed in the next section.

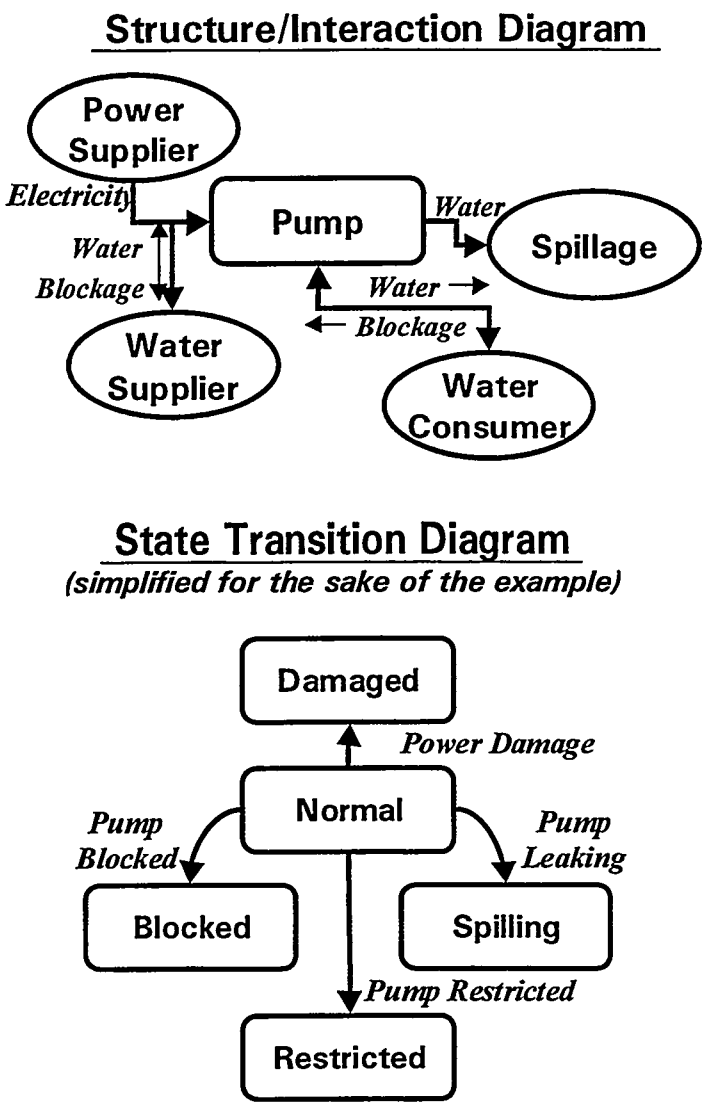


Figure 6-15. Simple Object Model for a Pump, Part 1.

The object model for a pump can be found in Figures 6-15 through 6-17. Figure 6-15 provides the structure and interaction diagrams for the pump object, as well as the state transition diagram. The reader will note that the interaction diagram for the pump is identical to that of the pipe except for the addition of a power supply.

The state transition diagram for the pump is also very similar to that of the pipe in that it contains normal, blocked, restricted, and spilling states. These are characteristic of a system that is designed to carry liquids. The simple pump model, however, contains an additional state related to the fact that the electrical side of the pump can become damaged—in this case, we assume by inappropriate electrical power, but other types of damage are also possible. Because the pump represents the translation of electrical energy into a flow in the liquid domain, it is reasonable to expect its object model to be more complex than that of a component that exists in only a single domain.

Also, for the sake of this analysis, we assumed that the pump is not repairable within the time frame of interest for the analysis. This assumption was made to simplify the example problem. Because of this assumption, the damaged, blocked, restricted, and spilling states in this model are all absorbing states. To convert this model to a repairable pump, one would have to include events to transition the model back from these states into the normal state based on the characteristics of a repair event.

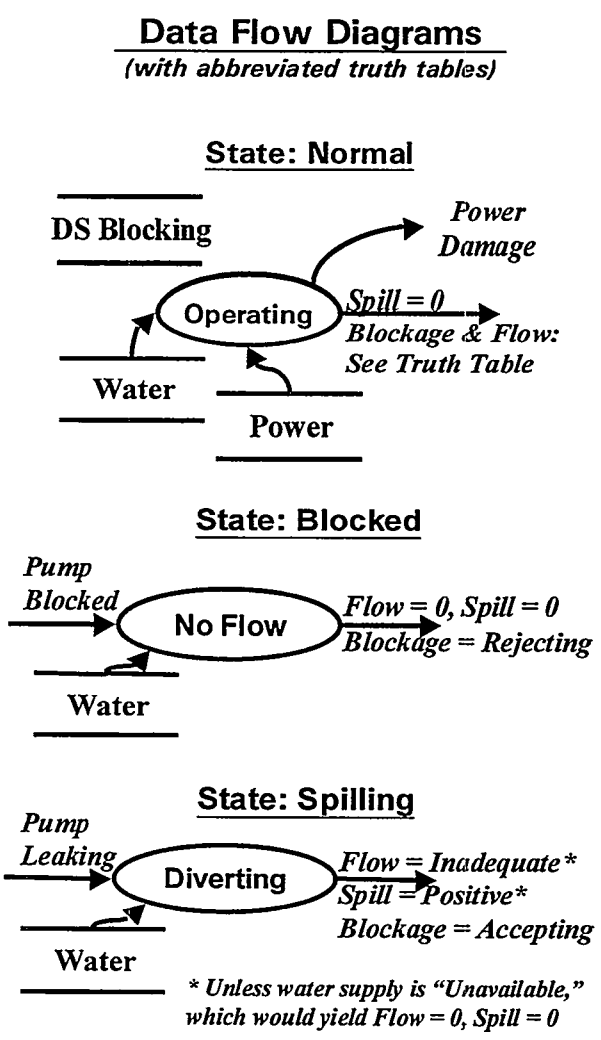


Figure 6-16. Simple Object Model for a Pump, Part 2

The data flow diagrams for most states in the pump object model look very similar to those used in the pipe object model. There is particularly close similarity in the models for the blocked and spilling states. In addition, the damaged state is assumed to behave very similarly to the blocked state in that the pump is assumed to prevent passage of liquid – not just restrict it.

The behavior of the pump in the normal and restricting states is significantly more complex. The truth table for the behavior of the pump in the operating transformation (in the normal state) is shown in Table 6-9. Here we see how various logical combinations of power availability, water supply availability, and downstream blockage combine to affect the output flow rate of the pump as well as the blockage it presents to upstream members. This truth table also describes how the pump object responds to inappropriate power – it generates the "power damage" event that causes the pump to transition into the damaged state. Obviously, a more complex pump model that included more levels of output flow rate would require a more complex truth table. While such a truth table would be difficult to populate by hand, it could be constructed once and placed in an object model library for later reuse in other analyses, thus lessening the analytical burden for later analysts.

Now that we have finished describing the generic object models for each of the physical components in our water supply system, it is worthwhile to go back to the original interaction diagram (Figure 6-10) and table of message values (Table 6-7) to see how these objects interact with one another. This system accepts only two events from the outside: the operator commands to turn the system on and off, which act on the system through the switch object. All other communication with outside entities is handled through the passing of discrete attribute values such as electrical power status, reservoir status, and blockage within the usage system. This system also does not generate events that are used by objects that are external to the system itself. It passes the results of its

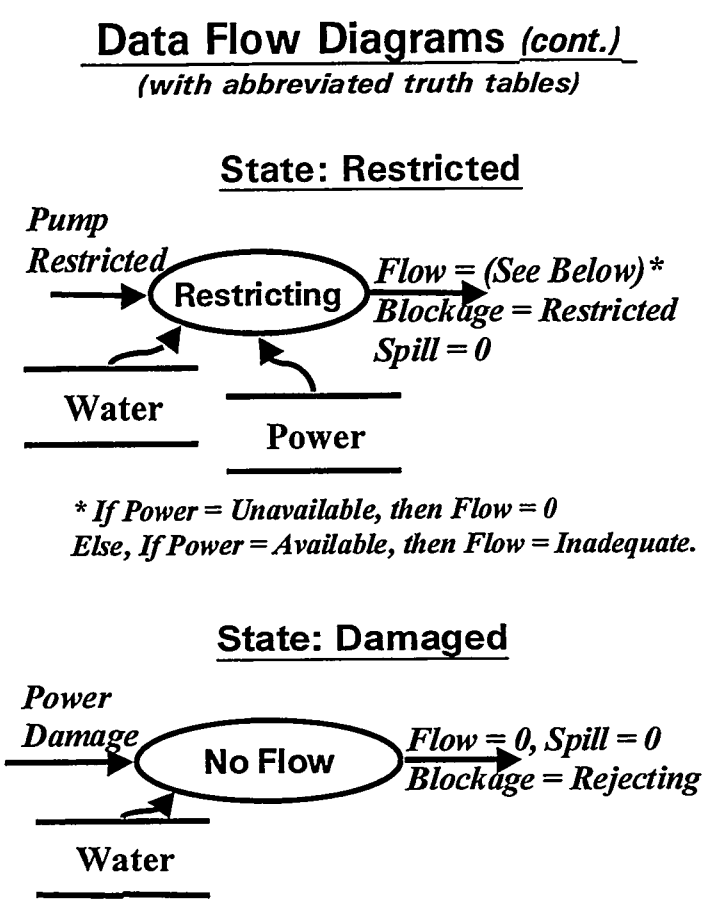


Figure 6-17. Simple Object Model for a Pump, Part 3.

own transformation to the outside world through the use of discrete attribute values for the spillage and for the flow rate of water to the usage system.

The reader should also be aware of the consistency requirements that must be imposed between objects if the overall collection of objects is to accurately represent a real system. In this sample problem, the pump, for example, has very specific expectations about the meaning of the power messages it receives from the wires, the switch, and the original power supply. If any of these objects has a different definition for "appropriate" or "inappropriate" power, this can cast serious doubt on the validity of the entire resulting object model and destroy the usefulness of any logic models derived therefrom. Therefore, in constructing an actual object model through the use of generic object components, the object construction and analysis software must verify two types of information: first, it must query the analyst for the specific characteristics of each instantiation of a generic component within the model (in order to customize the specific parameters of that instantiation), and second, it must check for consistency in the definition of messages that are to be passed between the various objects. In an early software implementation, these checks could be accomplished by querying the analyst. We believe that later and more complete implementations of this method will enable these consistency checks to be performed automatically within the analysis software, with only occasional queries to the user.

We have now completed the development of a common object model for the more detailed description of the water supply system. Again, the model is complete because each object's model is complete (as described at the end of Section 6.1), and because we have checked the consistency of events, attributes, and messages between objects. This common object model is now ready for use in a surety analysis.

Table 6-9. Data Flow Diagram for a Pump in the Normal State (a Truth Table for the Operating Transformation)

Power	Water Supply	Downstream Blockage	Event Output	Blockage	Output Flow Rate
X	Unavailable	X	—	Accepting	0
X	X	Rejecting	—	Rejecting	0
Unavailable	X	X	—	Rejecting	0
Inappropriate	X	X	Power damage	Rejecting	0
Available	Inadequate	Accepting	—	Accepting	Inadequate
Available	Inadequate or adequate	Restricting	—	Restricting	Inadequate
Available	Adequate	Accepting	—	Accepting	Adequate
X = Don't Care (Any value of this attribute satisfies the logical condition)					

6.5 Extracting Fault Trees from the Expanded Model

One of the major advantages of developing a common object model for a system is that one can extract many different surety analysis models from the single common object model and do so with assured consistency and relatively little effort (assuming that automation is available to interpret the common object model). Let us now examine the common object model developed in Section 6.4 to determine some of the types of surety models one might be able to extract from it.

The method for developing fault trees, as documented in Appendix A, dictates that one begin their development through the selection of the normal system state and the definition of valid states and conditions. Once this is accomplished, the analyst defines the objective of the fault tree analysis. This objective is written down in the form of a logical combination of discrete attribute values and/or states from within the common object model. In the simple object model developed in Section 6.1, there were very few discrete attribute values available to be part of a surety analysis objective. In the expanded water supply system model, there are many resultant and intermediate attributes that can be used as part of a fault tree objective statement. Consider once again the interaction diagram shown in Figure 6-10. For the original simple object model, one could ask questions about the delivery of water to the usage system or to the spillage system. In the expanded model, one can ask, not only about these final outcome attributes, but also about intermediate attributes such as the availability of water at the pump or at the outlet of pipe 1, or the availability of electricity at the output of the switch

or wire 1. Fault trees to examine such conditions are generally not done constructed because doing so is time and labor intensive. The ability to generate several different fault tree views of a system quickly and automatically can provide an analyst the opportunity to come to a far greater understanding of a system than would be possible using traditional analysis techniques.

One can also conduct an event tree analysis that examines the effect of specific random failure modes for specific components on the overall performance of a system. This is much better than what was possible in the simple model, where we were limited to just two random failure events – and those were assumed into the system without a basis in the actual system hardware. One could also conduct a failure modes and effects analysis by triggering the identified failure events one at a time, or a HAZOP analysis by varying the values of particular flows in the object model, again, one at a time. It is obvious, then, that this single common object model can support a wide variety of types of surety analyses, and that if software were available to automatically interrogate this common object model, these widely divergent surety analyses could be done quickly and with minimal additional effort. Contrast this with today's methodologies, which require a separate human-led model development effort for every type of analysis. In fact, in most cases, today one must engage in a separate human-led model development effort for each model instance (analysis objective) of each model type because of the difficulties and risks involved in sharing model fragments with unknown assumptions and pedigree.

Let us now extract one example fault tree from the extended common object model developed in Section 6.4. We will again focus on the condition where zero flow is available to the usage system in order to enable the reader to more directly compare this fault tree with the one developed from the simple common object model.

The first step in fault tree development, according to the rules in Appendix A, is to select the normal state for the system. In order to make this fault tree as comparable as possible with the previous examples, we will select a normal system state in which the overall water supply system is delivering water as requested to the usage system. This implies that all pipes are in the transmitting state, that all wires are in the conducting state, that the switch is in the OK, closed state, and the pump is in the normal state. The entities outside of the system are assumed to be behaving as follows: water is available from the reservoir; electricity is available and appropriate; the usage system is accepting water; and spillage of water from the system is zero. Given these conditions, the object model determines that the flow rate of water to the usage system is adequate.

The second step in fault tree development is to define the set of valid system states and conditions. For this system, there are no states that are either impossible or out of bounds as initial conditions or as intermediate conditions.

The third step in fault tree development is to define the objective of the fault tree analysis. In this case, the objective is to develop a fault tree that represents the ways the system can produce zero flow to the usage system. This condition manifests itself at the output of pipe 2 in the object model. Therefore, the top event of the fault tree is an OR gate that examines the delivery of water from pipe 2 in the water supply system.

The development of the fault tree continues according to the rules in Appendix A and follows a pattern that is very similar to that seen in the two example problems in Section 6.2. The actual fault tree that was developed according to these rules and for this condition can be found in eight tables beginning with Table 6–10. Note that this fault tree had to be broken into several sections (designated by part numbers in order to fit into this report. The fault tree printout is roughly organized according to a top-to-bottom flow of the major fault tree sections, with smaller repeated sections of the tree reserved for the end.

The fault tree begins by examining how pipe 2 can fail to provide flow – either in a typical (normal) state or in an abnormal state. The only abnormal state that can provide zero flow is the plugged state, which can only be entered as a result of a random failure event. The normal state (transmitting) can cause zero flow rate based on two lines in its truth table (see Table 6-8): in line 1, the water supply to the pipe is unavailable, and in line 2, the element downstream of the pipe is rejecting flow. Since the downstream system is the usage system, this branch of the fault tree consists simply of an external event indicating that the usage system is rejecting flow.

Consider now the water supply to pipe 2. This pipe is supplied with water only by the pump, and, on examining the object model for the pump, one concludes that the pump can provide zero flow either in its typical state (normal) or in two of its abnormal states (blocked or damaged). Entry into the blocked and damaged states ultimately occurs only as a result of state transitions, one of which is only caused by external random events; the other is caused by inappropriate electrical power. If we apply the rules from Appendix A, we trace back through the electrical components (wire 2, the switch, and wire 1) to find the fault subtree representing the methods by which inappropriate power can arrive at the pump. By consideration of the normal state, we determine that there are four different conditions (truth table lines 1 through 4 – see Table 6-9) that can lead to failure of the pump to supply water: the pump's water supply is unavailable, the pump is unable to deliver water because of a downstream blockage; the pump's power is unavailable; or the pump's power is inappropriate. The rules compel us to examine each of these four conditions – even though the inappropriate power branch will yield the same information that was found previously during examination of the damaged state. This is not a problem because the fault tree analysis software will sort out this duplication as part of its normal solution methodology.

The development of a fault tree model from the common object model produces an additional and somewhat unexpected set of duplicate conditions. This duplication comes about because of the behavior of the pipe and pump models in the common object model. Recall that these models are capable of not only indicating whether they are themselves blocked but also of transmitting to upstream components whether any downstream components are blocked. With this in mind, let us now consider pipe segment 1, the fault tree for which is found in Table 6–11. At first it seems as if everything is as expected, in that zero flow can occur in pipe segment 1 because the pipe is either in an abnormal state (plugged), or in the normal state (transmitting). In the normal state, two different conditions can cause zero flow. These are found in lines 1 and 2 of the truth table found

Table 6–10. Expanded Fault Tree for the Zero Flow Condition, Part 1

- FLOW-ZERO — The WSS flow rate is zero.
  - F0-P2-ABNORMAL — Zero flow rate achieved from a non-normal operating state.
    - P2-PLUG-STATE — Pipe 2 is in the plugged state.
  - F0-P2-TYPICAL — Zero flow rate achieved from typical system operating state.
    - F0-P2-TRANSMIT — Zero flow rate from pipe 2 in its typical transmitting state
      - F0-P2-TRAN-TRANS — Zero flow rate achieved in the transmitting transformation for pipe 2.
        - P2-TRANS-L2 — Conditions at the pipe 2 transmitting truth table line 2.
          - US-REJECT — External entity usage system attribute is rejecting.
        - P2-TRANS-L1 — Conditions at the pipe 2 transmitting truth table line 1.
          - P2-SUPP-0 — The water supply attribute to pipe 2 from pump has attribute unavailable (0).
            - F0-PUMP — Zero flow rate occurs at output of pump.
              - F0-PUMP-ABNORMAL — Zero flow rate from pump in a non-normal operating state.
                - PUMP-BLOCK-STATE — Zero flow achieved from pump in the blocked state.
                  - F0-PUMP-BNF-TRAN — Zero flow rate achieved in the pump B locked no flow transformation.
                    - P-BNF-TRANS-L1 — Conditions at the pump blocked no flow transformation truth table Line 1.
                - PUMP-DAMAG-STATE — Zero flow achieved from pump in the damaged state.
                  - F0-PUMP-DNF-TRAN — Zero flow rate achieved in the pump damaged no flow transformation.
                    - P-DNF-TRANS-L1 — Conditions at the pump damaged no flow transformation truth table line 1.
              - F0-PUMP-TYPICAL — Zero flow rate from pump in its typical normal state.
                - F0-PUMP-OP-TRANS — Zero flow rate achieved in the operating transformation for pump.
                  - P-OP-TRANS-L4 — Conditions at pump operating transformation line 4.
                  - P-OP-TRANS-L3 — Conditions at pump operating transformation line 3.
                  - P-OP-TRANS-L2 — Conditions at pump operating transformation line 2.
                  - P-OP-TRANS-L1 — Conditions at pump operating transformation line 1.
                    - PUMP-SUPP-0 — The water supply attribute to pump from pipe 1 has attribute unavailable (0).
                      - F0-P1 — Zero flow rate occurs at output of pipe 1.

Table 6–11. Expanded Fault Tree for the Zero Flow Condition, Part 2

△ F0-P1
└⌂ F0-P1 — Zero flow rate occurs at output of pipe 1.
  ├⌂ F0-P1-ABNORMAL — Zero flow rate from pipe 1 in a non-normal operating state.
  │ └⌂ P1-PLUG-STATE — Pipe P1 is in the plugged state.
  │ └⌂ P1-NOFLOW-TRANS — Zero flow rate achieve d in the no flow transformation.
  │ └⌂ P1-NOFLOW-L1 — Conditions at the pipe 1 no flow transformation truth table line 1
  │ └⌂ P1-PLUGGED — The external random event pipe plugged occurs for pipe 1.
  └⌂ F0-P1-TYPICAL — Zero flow rate from pipe 1 in its typical system operating state.
  └⌂ F0-P1-TRANSMIT — Zero flow rate from pipe #1 in its typical transmitting state.
  └⌂ F0-P1-TRAN-TRANS — Zero flow rate achieved in the transmitting transformation for pipe 1.
  ├⌂ P1-TRANS-L1 — Conditions at the pipe 1 transmitting truth table line 1.
  │ └⌂ P1-SUPP-0 — Water supply attribute to pipe 1 from reservoir has attribute unavailable (0).
  │ └⌂ RESERVR-UNAVAIL — External entity reservoir attribute is unavailable.
  └⌂ P1-TRANS-L2 — Conditions at the pipe 1 transmitting truth table line 2.
  └⌂ PUMP-REJECT — The Pump is object blockage attribute has value rejecting.
  ├⌂ REJ-PUMP-TYPICAL — Pump has attribute rejecting in its typical normal operating state.
  │ └⌂ REJ-PUMP-OP-TRAN — Rejecting is achieved in the operating transformation for the pump.
  │ ├△⌂ P-OP-TRANS-L2 — Conditions at pump operating transformation line 2.
  │ ├△⌂ P-OP-TRANS-L3 — Conditions at pump operating transformation line 3.
  │ └△⌂ P-OP-TRANS-L4 — Conditions at pump operating transformation line 4.
  └⌂ REJ-PUMP-ABNORM — Pump has attribute rejecting in a non-normal operating state.
  ├⌂ REJ-PUMP-BLOK-ST — Pump has attribute rejecting in the blocked state.
  │ └⌂ REJ-PUMP-BNF-TRA — Rejecting attribute set in the pump blocked no flow transformation.
  │ └△⌂ P-BNF-TRANS-L1 — Conditions at the pump blocked no flow transformation truth table line 1.
  └⌂ REJ-PUMP-DAM-ST — Pump has attribute rejecting in the damaged state.
  └⌂ REJ-PUMP-DNF-TRA — Rejecting attribute set in the pump damaged no flow transformation.
  └△⌂ P-DNF-TRANS-L1 — Conditions at the pump Damaged No Flow Transformation truth table Line 1.

Table 6–12. Expanded Fault Tree for the Zero Flow Condition, Part 3

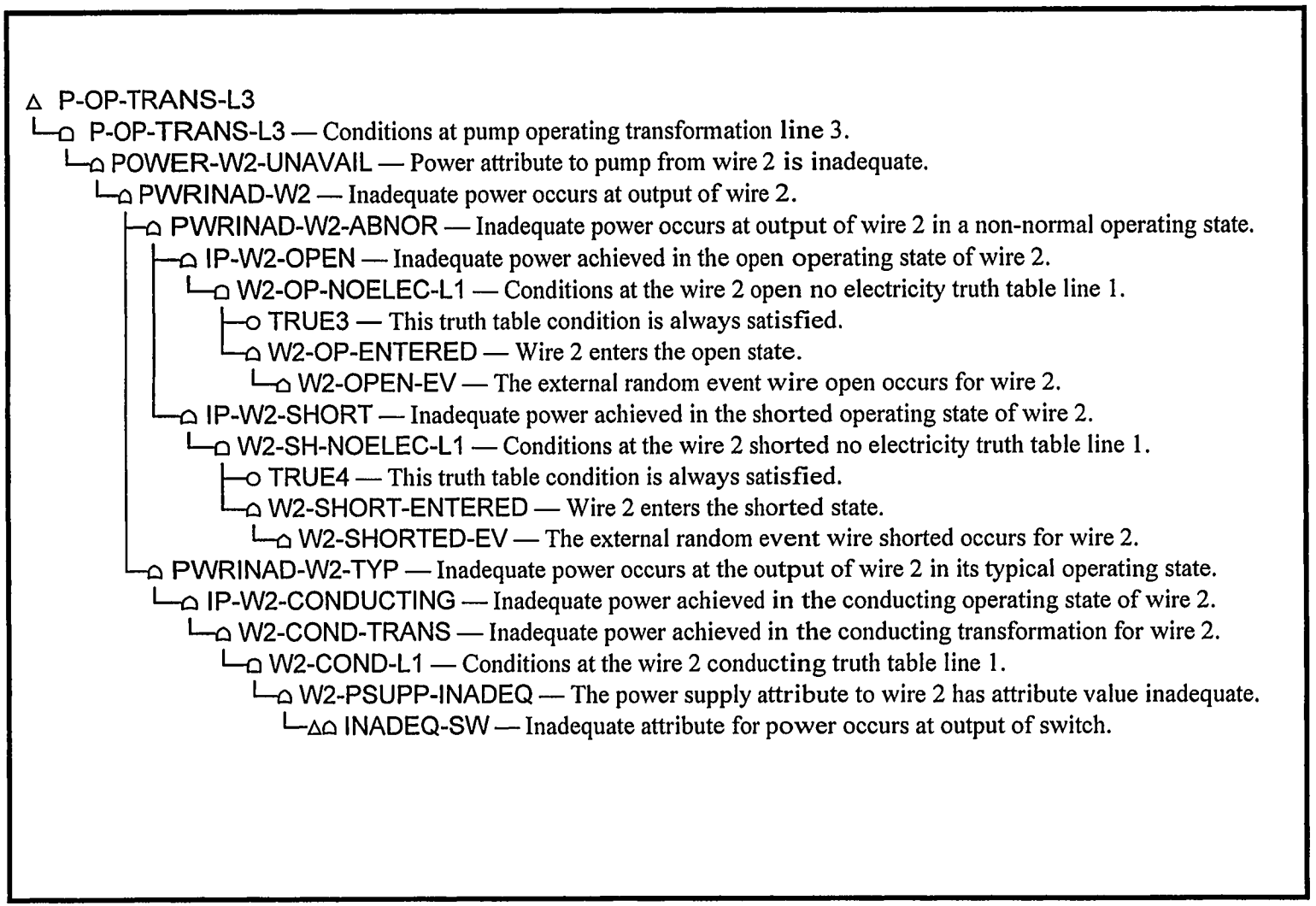


Table 6–13. Expanded Fault Tree for the Zero Flow Condition, Part 4

△ INADEQ-SW
└⌂ INADEQ-SW — Inadequate attribute for power occurs at output of switch.
  ├⌂ INAD-SW-ABNORMAL — Inadequate power occurs at the output of switch in an abnormal operating state.
  │ ├⌂ INAD-SW-OKOPEN — Inadequate power achieved in the OK open operating state of switch.
  │ │ └⌂ IP-SW-OKOP-NOELE — Inadequate power achieved in OK open no electricity transformation for switch.
  │ │ └⌂ SW-OKOP-NOELE-L1 — Switch OK open no electricity transformation truth table Line 1.
  │ │ ├○ TRUE1 — This truth table condition is always satisfied.
  │ │ └⌂ SW-OKOP-ENTERED — Switch OK open state entered.
  │ │ └⌂ OFF — External entity operator issues the off event.
  │ └⌂ INAD-SW-FOPEN — Inadequate power achieved in the fail open operating state of switch.
  │ └⌂ IP-SW-FOP-NOELEC — Inadequate power achieved in fail open no electricity transformation for switch.
  │ └⌂ SW-FOP-NOELE-L1 — Switch fails open no electricity transformation truth table Line 1.
  │ ├○ TRUE2 — This truth table condition is always satisfied.
  │ └⌂ SW-FOP-ENTERED — Switch fail open state entered.
  │ ├⌂ SW-OFTC — Random external event switch open, fails to close occurs.
  │ └⌂ SW-CFTRC — Random external event switch closed, fails to remain closed occurs.
  └△⌂ INAD-SW-TYPICAL — Inadequate power occurs at the output of switch in its typical operating state.

Table 6–14. Expanded Fault Tree for the Zero Flow Condition, Part 5

△ INAD-SW-TYPICAL
- INAD-SW-TYPICAL — Inadequate power occurs at the output of switch in its typical operating state.
  - INAD-SW-OKCLOSED — Inadequate power achieved in the OK closed operating state of switch.
    - IP-SW-CONDUCTING — Inadequate power achieved in the conducting transformation for switch.
      - SW-COND-L1 — Conditions in the switch conducting truth table line 1.
        - SW-PSUPP-INADEQ — The power supply attribute to switch has attribute value Inadequate.
          - INADEQ-W1 — Inadequate attribute for power occurs at output of wire 1.
            - INAD-W1-TYPICAL — Inadequate power occurs at the output of wire 1 in its typical operating state.
              - IP-W1-CONDUCTING — Inadequate power achieved in the conducting operating state of wire 1.
                - W1-COND-TRANS — Inadequate power achieved in the conducting transformation for wire 1.
                  - W1-COND-L1 — Conditions at the wire 1 conducting truth table line 1.
                    - W1-PSUPP-INADEQ — The power supply attribute to wire 1 has attribute value inadequate.
                      - PWR-INADEQ — External entity power supply attribute is inadequate.
            - PWRINAD-W1-ABNOR — Inadequate power occurs at output of wire 1 in a non-normal operating state.
              - IP-W1-OPEN — Inadequate power achieved in the open operating state of Wire 1.
                - W1-OP-NOELEC-L1 — Conditions at the wire 1 open No Electricity truth table Line 1.
                  - TRUE5 — This truth table condition is always satisfied.
                  - W1-OP-ENTERED — Wire 1 enters the open state.
                    - W1-OPEN-EV — The external random event wire open occurs for wire 1.
              - IP-W1-SHORT — Inadequate power achieved in the shorted operating state of wire 1.
                - W1-SH-NOELEC-L1 — Conditions at the wire 1 shorted no electricity truth table line 1.
                  - TRUE6 — This truth table condition is always satisfied.
                  - W1-SHORT-ENTERED — Wire 1 enters the shorted state.
                    - W1-SHORTED-EV — The external random event wire shorted occurs for wire 1.

Table 6–15. Expanded Fault Tree for the Zero Flow Condition, Part 6

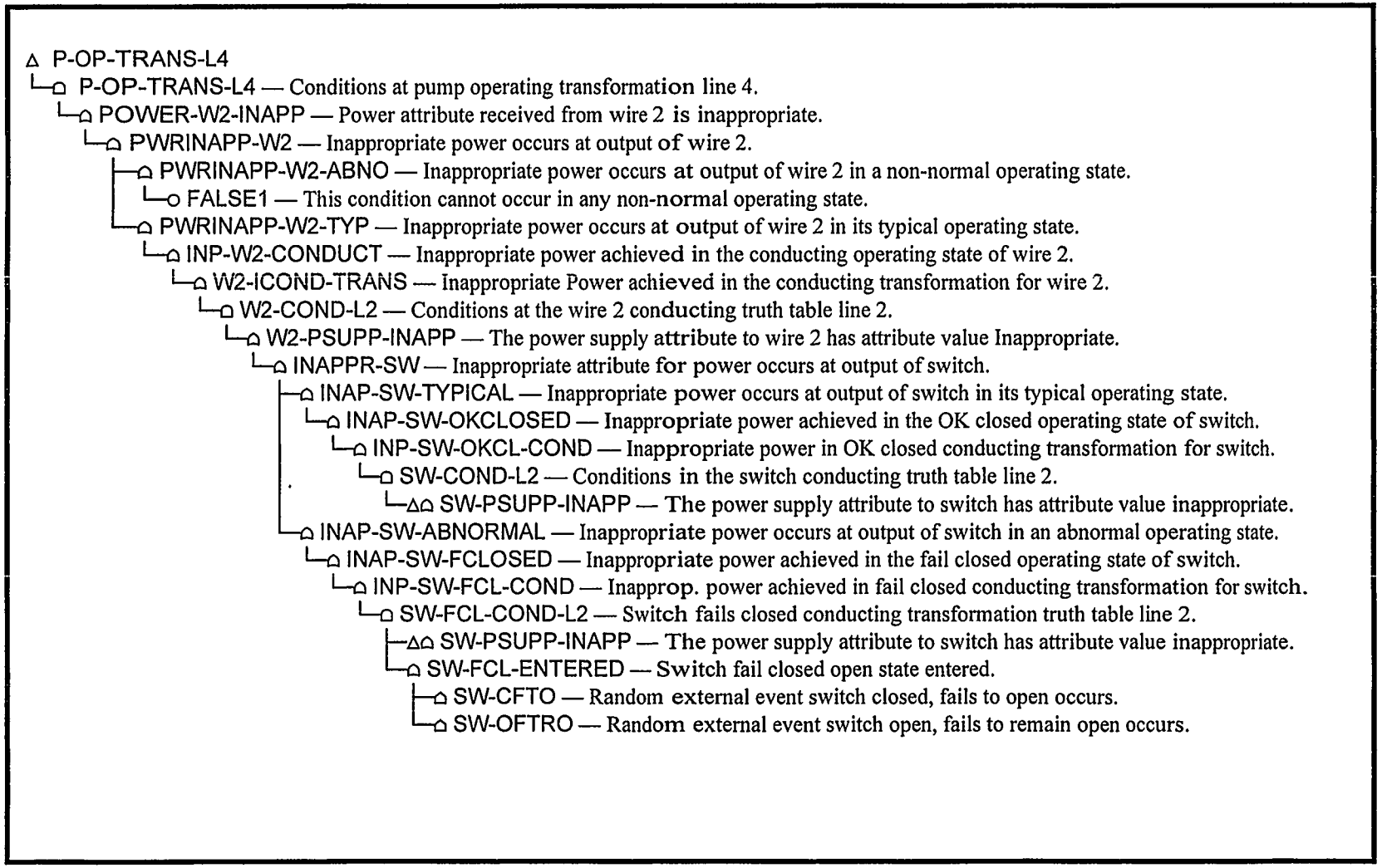


Table 6–16. Expanded Fault Tree for the Zero Flow Condition, Part 7

△ SW-PSUPP-INAPP
└⌂ SW-PSUPP-INAPP — The power supply attribute to switch has attribute value Inappropriate.
  └⌂ PWRINAPP-W1 — Inappropriate attribute for power occurs at output of wire 1.
    ├⌂ PWRINAPP-W1-ABNO — Inappropriate power occurs at output of wire 1 in a non-normal operating state.
    │ └○ FALSE2 — This condition cannot occur in any non-normal operating state.
    └⌂ PWRINAP-W1-TYP — Inappropriate power occurs at output of wire 1 in its typical operating state.
      └⌂ INP-W1-CONDUCT — Inappropriate power achieved in the conducting operating state of wire 1.
        └⌂ W1-ICOND-TRANS — Inappropriate power achieved in the conducting transformation for wire 1.
          └⌂ W1-COND-L2 — Conditions at the wire 1 conducting truth table line 2.
            └⌂ W1-PSUPP-INAPP — The power supply attribute to wire 1 has attribute value inappropriate.
              └⌂ PWR-INAPP — External entity power supply attribute is inappropriate.

△ P-OP-TRANS-L2
└⌂ P-OP-TRANS-L2 — Conditions at pump operating transformation line 2.
  └⌂ P2-BLK-REJECTING — Blockage attribute from pipe 2 has value rejecting.
    └△⌂ P2-PLUG-STATE — Pipe 2 is in the plugged state.

△ P2-PLUG-STATE
└⌂ P2-PLUG-STATE — Pipe 2 is in the plugged state.
  └⌂ P2-NOFLOW-TRANS — Zero flow rate achieved in the no flow transformation.
    └⌂ P2-NOFLOW-L1 — Conditions at the pipe 2 no flow transformation truth table line 1
      └⌂ P2-PLUGGED — The external random event pipe plugged occurs for pipe 2.

Table 6-17. Expanded Fault Tree for the Zero Flow Condition, Part 8.

△ P-BNF-TRANS-L1
└⌂ P-BNF-TRANS-L1 — Conditions at the pump blocked no flow transformation truth table line 1.
  ├○ TRUE8 — This truth table condition is always satisfied.
  └⌂ P-BLK-ENTERED — The pump enters the blocked state.
    └⌂ PUMP-BLOCKED — The external random event pump blocked occurs.

△ P-DNF-TRANS-L1
└⌂ P-DNF-TRANS-L1 — Conditions at the pump damaged no flow transformation truth table line 1.
  ├○ TRUE7 — This truth table condition is always satisfied.
  └⌂ P-DAM-ENTERED — The pump enters the damaged state.
    └⌂ PWRDAM-EVENT — The power damage event occurs.
      └⌂ PDAM-EV-PUMP — Power damage event is generated by the pump object.
        └⌂ PDAM-EV-PNORMAL — The power damage event is generated by the pump when it is in the normal state.
          └⌂ PD-PUMP-OP-TRANS — Power damage event occurs from the operating transformation for the pump.
            └△ P-OP-TRANS-L4 — Conditions at pump operating transformation line 4.

in Table 6-8. As expected, one of these lines indicates that zero flow will occur if the water supply to pipe segment 1 is unavailable. This is easily traced back to an external event condition where the reservoir is unavailable. The second line of that truth table, however, yields the unexpected results. It indicates that the flow rates through pipe segment 1 will be zero if one or more of the entities that are downstream are rejecting flow. The rules from Appendix A then push us to examine whether the pump is rejecting flow, which in turn causes us to examine whether pipe segment 2 is rejecting flow, and ultimately whether the usage system is rejecting flow. Recall that the usage system rejecting flow was considered at the very beginning of the fault tree. If one were developing a fault tree by hand, one would intuitively understand that these conditions had already been examined in other areas of the fault tree and thus neglect them. However, the "automated" construction of the fault tree from the common object model caused this correct but unnecessary section of fault tree to be constructed. Again, duplicate sections of a fault tree are not a problem (as long as they are not contradictory) because the fault tree analysis software will sort out the duplication as part of its normal solution methodology.

Let us examine this apparently duplicated section of the fault tree more closely and see whether it is in fact unnecessary in all circumstances for the fault tree construction engine to examine downstream blockages. Consider the situation where one is seeking to build a fault tree that would examine the reasons for zero flow at the outlet of pipe segment 1. One reason for zero flow at that point would, in reality, be the presence of downstream blockages in the water supply system. Thus, if we programmed the fault tree construction engine to neglect these downstream blockages (and other possibly similar conditions), we would have missed a potentially important section of the fault tree for this other situation. Therefore, it is not advisable to seek situations where fault tree truncation might be employed during the fault tree construction process because this can lead to situations where important parts of another fault tree are inadvertently neglected. Furthermore, there is no sure way of knowing from the results of the fault tree analysis whether a portion of the fault tree was neglected. While it is never good to obtain incomplete results in a surety analysis, it is even worse to obtain incomplete results without having any indication that those results might be incomplete, as would be true in this case.

This point brings us back to one of the reasons why analysts do not have much confidence in reusing pieces of fault trees that were developed by other analysts for other purposes. One can never be entirely sure of all of the assumptions that went into the production of the original fault tree, and without knowing those assumptions, it is impossible to determine whether a new application of that fault tree segment will violate them. In fact, the original analyst may not have even realized that he was making some of his assumptions, so simply asking the original analyst to document his assumptions will not have the desired results. However, re-deriving the fault trees from an object-based model that incorporates the behaviors and causes and effects within the system alleviates this problem because the object-based model makes all of these assumptions explicit.

It should be noted at this point that the fault tree development rules specified in Appendix A may fail under certain specific circumstances because they do not make explicit provision for breaking "logic loops." A logic loop occurs when one seeks to build a fault tree for two systems that are mutually interdependent. For example, if a diesel-powered emergency electrical generator is cooled by an electrically powered cooling system, and that cooling system draws its power from the emergency diesel generator, these two systems are mutually interdependent. The generator can fail because the cooling system fails, but the cooling system can fail because the generator fails. Such a condition will drive the fault tree development rules from Appendix A into an infinite loop condition. It is relatively easy for a human to break such logical loops, and automated processes are available. However, the fact that such logic was not incorporated into the rules in Appendix A is an important limitation for that methodology that must be observed.

There is one common objection to the fault tree models produced by extraction from the object model: they are much larger than a similar fault tree constructed by a human being would be. The large size of the fault trees is in large part due to the presence of many single input gates. This is, in fact, a valid criticism. However, it is important to realize that this fault tree construction methodology thoroughly documents every small step in logic as a gate in the fault tree. It is, in a sense, practicing the oft-preached art of "immediate cause" in its fault tree development.[1] The net result is a fault tree that is easy to trace because it contains documentation for every small logical step. While such a tree is large, it presents only a minimal additional challenge to a fault tree solution software package because such packages typically restructure fault trees to remove these unnecessary gates before embarking on the ultimate solution. The fault tree restructuring process represents only a minimal additional computational burden when it is compared with the effort required to actually solve the fault tree. While such fault trees can be burdensome to print, the documentation detail they contain provides valuable insight into the construction process that can help the surety analyst check the validity of the resultant fault tree.

6.6 Summary

In this chapter we have constructed two common object models for a simple water supply system: a simple functional model and a component-based physical model. We demonstrated how the rules provided in Appendix A can be applied to the common object models to extract a variety of fault trees, event trees, and other surety models. We explicitly extracted three fault trees and two event trees from the various object models. We also discussed how the fault trees and event trees derived using these methods might be somewhat different from those that would be generated by a human analyst. Provided the causality and behavior embodied in the common object model are correct and complete, the surety models that are automatically extracted from that common object model should be more complete (or be based on fewer assumptions) than would a comparable human-generated surety model. This is because the rules for automatic model extraction intentionally examine all possible areas of the common object model, while the human analyst will often truncate portions of the model that are unimportant for the particular purposes of the analysis. While this usually leads to acceptable results for

the surety analysis at hand, the presence of undocumented assumptions can make the reuse of model fragments a difficult and chancy proposition.

The example problems constructed in this chapter should also provide an illustration of how one might go about using this methodology in the presence of a fully functional software tool. We described the vision of reusable generic component libraries that will allow the rapid construction of the common object models. We also described the ease and speed with which multiple surety models can be derived, and how the surety analyst can use these various views of the system to obtain a better and more complete understanding of the system being analyzed. Alternatively, the analyst should be able to derive the same models that are currently being used much more quickly, resulting in cost savings. Either way, this methodology provides a valuable step forward in the techniques for performing surety analyses.

6.7 References

1. Roberts, N.H., Vesely, W.E., Haasl, D.F., and Goldberg, F.F., Fault Tree Handbook, NUREG-0492, U.S. Nuclear Regulatory Commission, Washington, DC, 1981.

7 Implementation in Software

One of the goals of this project was to develop a software application that would demonstrate the *methodology* developed by the project that applies risk and reliability analysis to object-oriented (OO) modeling of systems. A goal of the software development was to use emerging OO software technologies to implement the application. Early ideas were to provide a distributed architecture for the tool, using an object-oriented database (OODB) server, the common object request broker architecture (CORBA) for distribution, and the Java programming language to implement a client user application. The software development efforts for the first year of the LDRD and part of the second concentrated on learning these technologies and implementing prototypes to prove their feasibility. Indeed a prototype was developed that demonstrated a Java client application reading and writing objects to a Versant OODB via CORBA. As development of the methodology continued and time passed, it became clear that the development of the tool needed to be reevaluated. In early 1999, this calendar year it was decided to implement the tool as a stand-alone application on a PC platform without using a database. The application would still use Java, and be designed with components that could easily be migrated to create a distributed OODB application in the future. The desire was to create an application that could demonstrate the methodology in a manner that could be extended as needed. The scope of the application was redefined in ways that were felt would still meet the primary goals and make them achievable. It is believed that this effort was successful and has resulted in an application that is called the object-oriented process for risk and reliability analysis (OPRRA).

There are many commercial and custom software applications available that could have been used to create the tool. However, none of the tools we identified provided all of the functionality that was needed. In addition, it was not realistic to attempt to create a single application from scratch that would accomplish the goals. It was decided to develop an application that provided the *glue* to bring several existing applications together for the desired functionality. With limited time and resources to develop the tool, not a great deal of time was spent examining various options for the development. Rather, decisions were made fairly quickly on key features of the architecture of the tool, based on the perception of whether it would be successful.

The most important component needed was an application that would enter the OO model data for the system to be analyzed. There are a number of commercial applications available that are primarily designed for software development. This was a stumbling block in that there were some notions developed for the system modeling that did not translate well into the software development paradigm. Nevertheless, with some compromises, one of these tools was adopted.

One of the key features of the methodology is to use action data flow diagrams (DFD) to model the process for each state of the system. This was lacking in the available commercial applications.

Access to the OO data captured in an application was needed to translate it into a form usable for risk and reliability analysis such as fault trees, event trees, etc. Since a commercial tool was to be used to enter the OO data, it had to have a comprehensive application programming interface (API).

Rational Rose is one of the premier PC-based applications for object-oriented software development. Rational Rose 98 was released in early 1998. It did not provide a way to capture DFDs, but it did provide much of the other functionality that was needed. Consequently, it was chosen for our development.

The OPRRA application interfaces to Rose and can retrieve data that are stored in Rose as needed. OPRRA provides a graphical user interface (GUI) to enter the DFD for the states defined in Rose. Software has been developed to generate fault trees. Data from Rose and the OPRRA are used to create the fault tree. New risk and reliability analysis methods could easily be added to OPRRA. The fault trees generated by OPRRA are written to a file in a format that can be used by the ArrTree application. ArrTree is an application used to manually create fault trees that can be further processed by other existing Risk and Reliability Analysis software. These data can be processed by other applications for additional analysis.

7.1 Application Overview

Rational Rose 98 is used for the primary OO modeling of the system. Established practices are used to model the system in the OO paradigm. Rose supports several different modeling notations. We chose to use the unified modeling language that is rapidly being accepted as a standard notation.

Figure 7-1 shows the Rose screen with class diagram and state diagram windows. The diagrams show a simple system model that was used during development. The Controller class and the WaterSupplySystem class define two system objects. The other classes shown are defined to provide further information to support the methodology. Notice that the *power* attribute of WaterSupplySystem has a *type* of PowerType. The PowerType class is defined in the model with the attributes APPROPRIATE, INADEQUATE, and INAPPROPRIATE. These define the valid values for the *power* attribute. The attributes of PowerType also have a *type* of DAVType. The DAVType shows two attributes, NORMAL and ABNORMAL. The attributes of PowerType are declared with an *initial value* that is one of these attributes. The *initial value* is used during the processing to produce a fault tree. The remaining attributes for the system classes are defined in a similar manner.

State models are defined for the two system objects. The state diagram for the Controller class shows two states and the one for the WaterSupplySystem class shows three states. The arrows define transitions between the states. Notice that the transitions in the Controller state diagram are also labeled with a *send action*. The send action denotes that the transition affects a transition in the state machine of another class. For example, the On transition of the Controller state machine has a send action with the WaterSupplySystem as the target class and On as the target transition. The

WaterSupplySystem state diagram includes a transition named On. The state machine for WaterSupplySystem transitions between the IdleFunctional and DeliveryingAsRequested states when the On send action (i.e., event) is received.

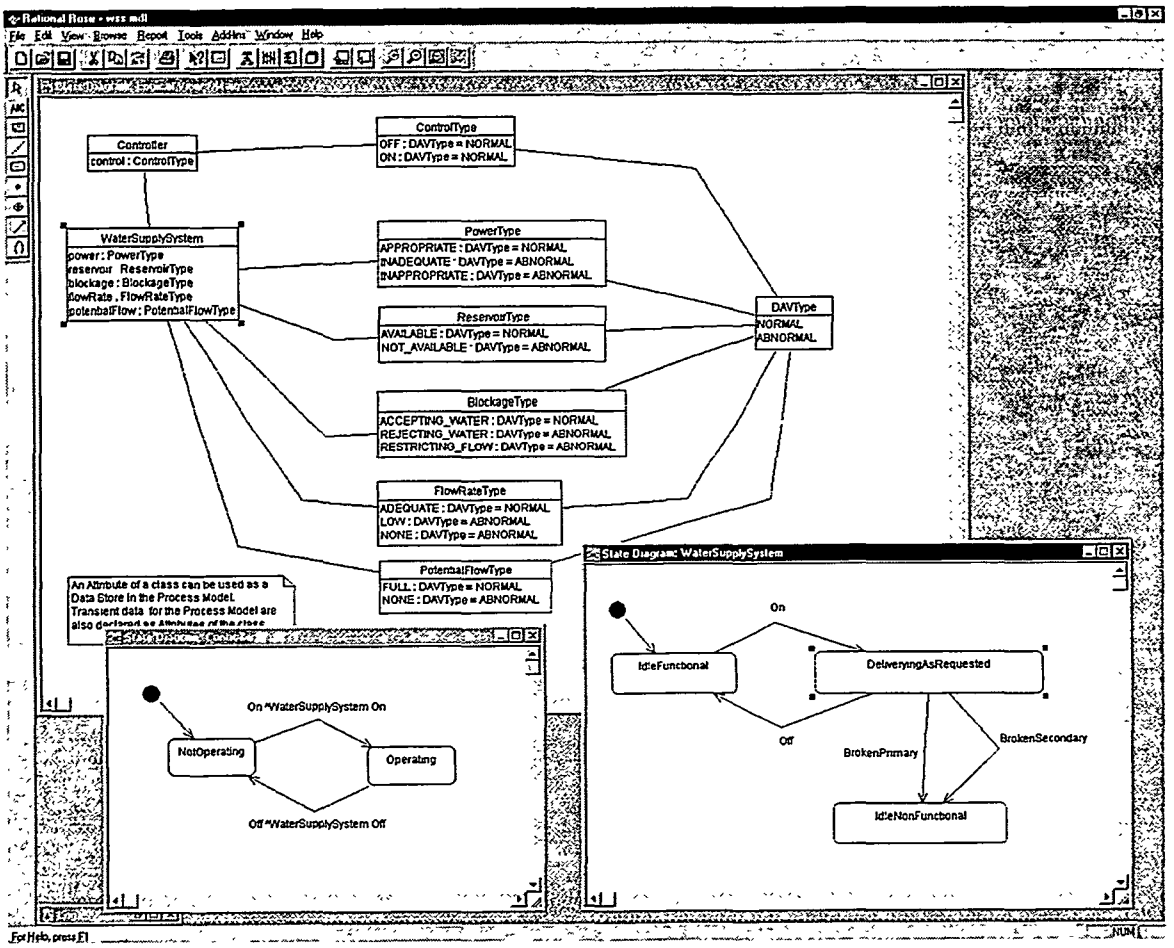


Figure 7-1. Rational Rose 98 Class and State Diagrams

Rose provides a way to extend its own menus with custom menus. A registry entry is made on the user's PC to specify a custom menu file. The menu file is created using a Rose-specific syntax. A menu file has been created for the OPRRA application so that it can be launched from Rose via the *Tools* menu.

Figure 7-2 shows a screen from the OPRRA application. It shows the process models that were created for the Rose model shown in Figure 7-1. Notice that there is a separate window for each process model, one for each state in the Rose model. The process model is defined in terms of a DFD. DFDs are usually represented graphically as bubbles, arrows, and boxes. However, in order to develop the application more quickly, it was decided to represent the DFD textually in a table. Graphical representation of the

DFD could be added in the future and should not change the underlying data structures. It is probably easiest for a user to manually create DFD diagrams and then enter the corresponding data in the OPRRA tables.

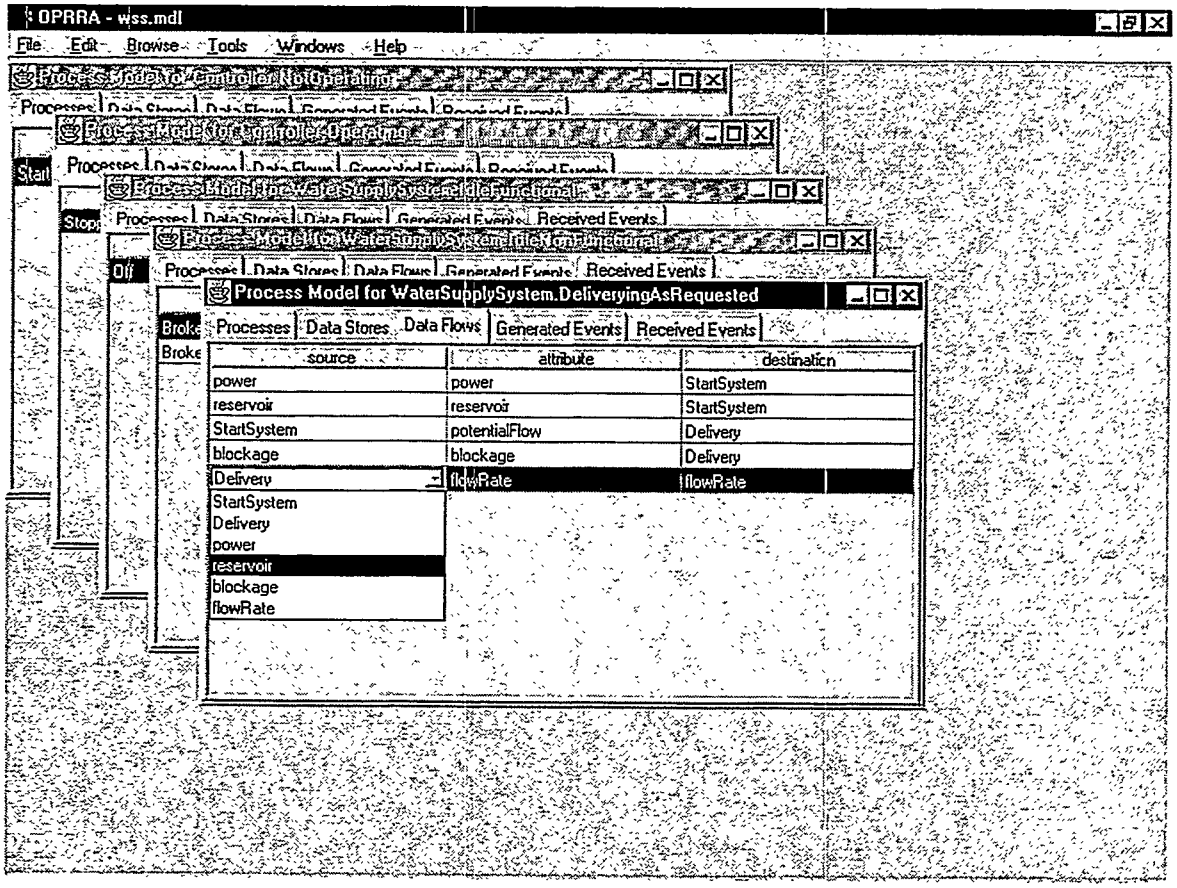


Figure 7-2. OPRRA Process Models.

The five main components of a DFD are process, data store, data flow, received event, and generated event.[1] A process contains a description of how data *input* to the process is transformed into data *output* from the process. The way this is done for our method is unique and is described later in the document. The attributes of a class are available to supply or receive data from a process. A data store is declared with a specified attribute, to make it available to the process. A data flow is defined from a data store to a process to show that the data store is an input to the process. The direction of the data flow indicates whether the data store is an input or output to the process. A data flow may also go from one process to another process. In this case the data flow *carries* a specified attribute that is used as a variable between the two processes. A received event contains a transition (one coming into the state for the process model) and the process it is destined for. Likewise, a generated event contains a transition (one leaving the state for this process model) and the process it is coming from.

List boxes are provided for data entry that give the user the valid selections for a particular entry. Selections may come from data previously entered in OPRRA or in Rose. For example, the figure shows a list of data flows for the deliveryingAs-Requested state of the WaterSupplySystem class. The source of a data flow may be either a process or a data store. The list box shows all of the processes and data stores that are defined for this process model. Note that the user switches between the DFD elements by selecting a tab.

For software development, some form of pseudo-code is typically used to describe a process. This is too loosely structured for our purposes, so the process description is defined in terms of a process table. Each line of the process table is referred to as a statement and the table is made up of a collection of statements. The inputs to the Process table are the attributes of data stores connected to the process by data flows coming into the process. The outputs of the process table are the attributes of data stores connected to the process by data flows going out of the process. Each input and output attribute is edited by selecting from a list of valid values. A statement is defined by specifying a combination of input attribute values that result in a set of output attribute values. A generated event may also be specified for each statement.

Figure 7-3 shows the dialog box used to enter the data for a process table. This box is presented when the user selects a process row and selects *Browse, Specification,* from the menu. The *Insert* button is used to create new rows in the table and the *Delete* and *Delete All* buttons remove rows. The columns for the table are predetermined by the data flows that are inputs and outputs to the process. Since there are no generated events for this process, a column for a generated event is not shown in the table.

When the user is satisfied that all model data are complete, a fault tree can be generated. The user specifies the *top event* for the fault tree. This is expressed in the form of one or more attribute value pairs. The user can specify whether the attribute should be equal to the value or whether it should not be equal. A fault tree is generated, starting at the specified class, by a procedure centered on the process table. Data flows may be followed from one process to another to develop branches of the fault tree. A received event may come from another class in the model, in which case it is followed *back* to that class to develop branches of the fault tree for that class as well. Multiple fault trees can be generated for the same system model by specifying a new top event. The fault tree data are written to two files (*.set* and *.dct* files) that make the data compatible with the ArrTree application.[2]

Figure 7-4 shows the ArrTree application display of a fault tree that was generated by OPRRA. The result is rather verbose and not immediately obvious. A gate in the fault tree marks each step of the fault tree generation. This results in extraneous gates but allows the user to validate the fault tree. The extra gates can be automatically removed by subsequent analysis software. Also, the fault tree is stated strictly in terms of the system object model and consequently may not be as readable as a fault tree manually generated by a human.

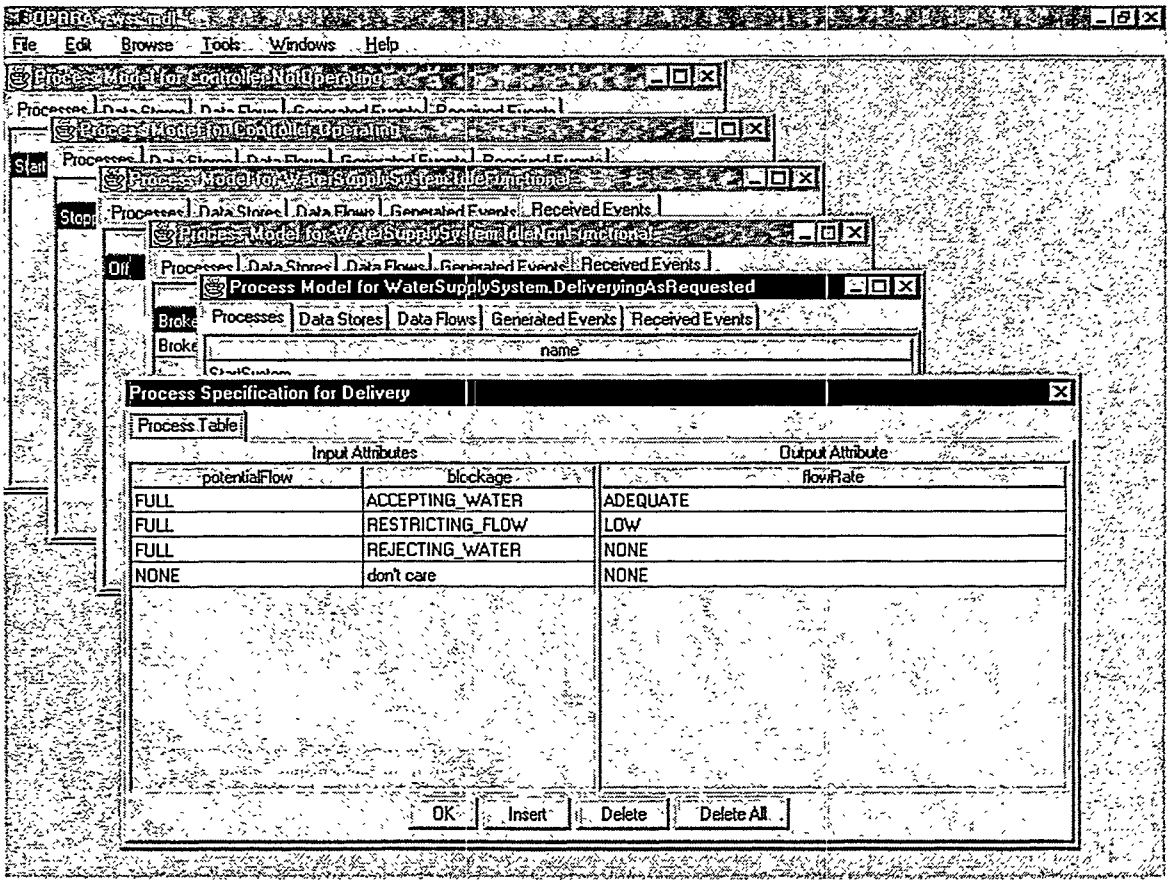


Figure 7-3. OPRRA Process Specification

7.2 Open Standards

In the future, software applications will be much more flexible in how they are used individually and with other applications. The beginning of this can be seen now in a number of desktop applications. This trend should continue so that a user can configure *custom* tools from components of existing tools, easily creating and adding new components as needed. In this respect Java provides some fundamental features that enable this.

Java classes are packaged in a manner that makes them easy to reuse in other applications. A Java application is *not* compiled into one homogeneous application, but exists as the set of classes used to create it. Each class is individually accessible for use by an external application, to the extent that the class and class methods are exposed.

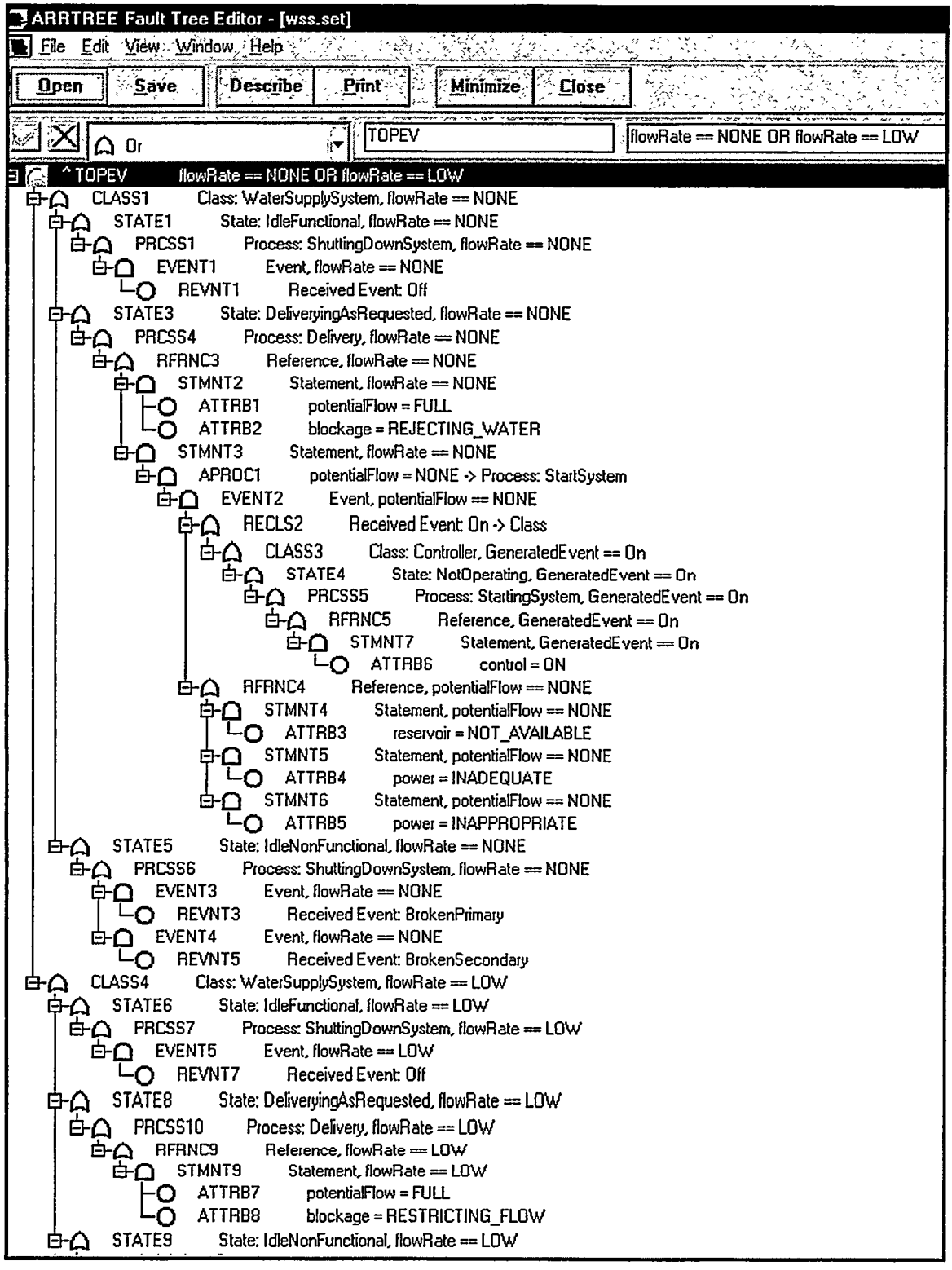


Figure 7-4. ArrTree Application Fault Tree Display

7.3 Overview of OPRRA Application Software Design

The OPRRA application software has been developed using OO analysis and design methods. The application exists as a collection of more or less independent classes that work together to perform the necessary functions. The classes can be described by dividing them into four categories. The main category contains the classes that provide the basic framework for the application. The Rose API interface category contains the classes that interface to the Rose API. The process model category contains the classes that capture the DFD for a process. The fault tree support category contains classes used in generation of fault trees from the system model data.

7.3.1 Main Category

The main category consists of two classes named OPRRA and Desktop. The OPRRA class contains the static *main* method that is called by the Java virtual machine to start the application. The main method instantiates the OPRRA class, which starts up the application displaying the main window and menus. An instance of the RApplication class is created that establishes a connection to the Rose application. If Rose is not already running, a new application is started. Classes from other categories are instantiated as needed to support menu selections.

The OPRRA class provides methods that save and load data entered in the application. Data are saved using class *serialization.* Serialization is a *built-in* feature of Java classes that allows instances of a class to be written to an output stream, typically a file. The structure of the class is saved as well as the data so that a class instance can be recreated when it is *de-serialized.* When a class is serialized, all other classes that are referenced by the class are also serialized. This is done in an intelligent way so that a class is only serialized once even though it may be referenced multiple times by other classes.

The Desktop class is serialized to save all of the OPRRA application data. OPRRA is designed so that all of the classes that should be saved are children of Desktop. This includes the classes that capture the process model data as well as the GUI components that display the data. When Desktop is *re-loaded,* the data are restored and displayed in the same manner as when they were saved.

7.3.2 Rose API Interface Category

The Rose application provides a comprehensive API that is called the Rose extensibility interface (REI). The REI can be used with a scripting language or in the Microsoft *OLE automation* environment. OPRRA uses the automation environment. The components of Rose made available by the REI are organized in an OO manner. Figure 7-5 shows a class diagram of the Rose REI. Each box in the diagram represents an interface that can be used to manipulate the corresponding object in Rose. Typically each interface has a set of properties and methods that can be used for this control. Properties reference the *attributes* of the object that can be written and read by standard property methods. A method of the interface performs a process on the object. An inheritance hierarchy is

used. For instance, many objects inherit from the element object. The element object has a *name* attribute that is commonly used to identify instances of an object in Rose. Different types of objects have this attribute since they inherit *name* from element.

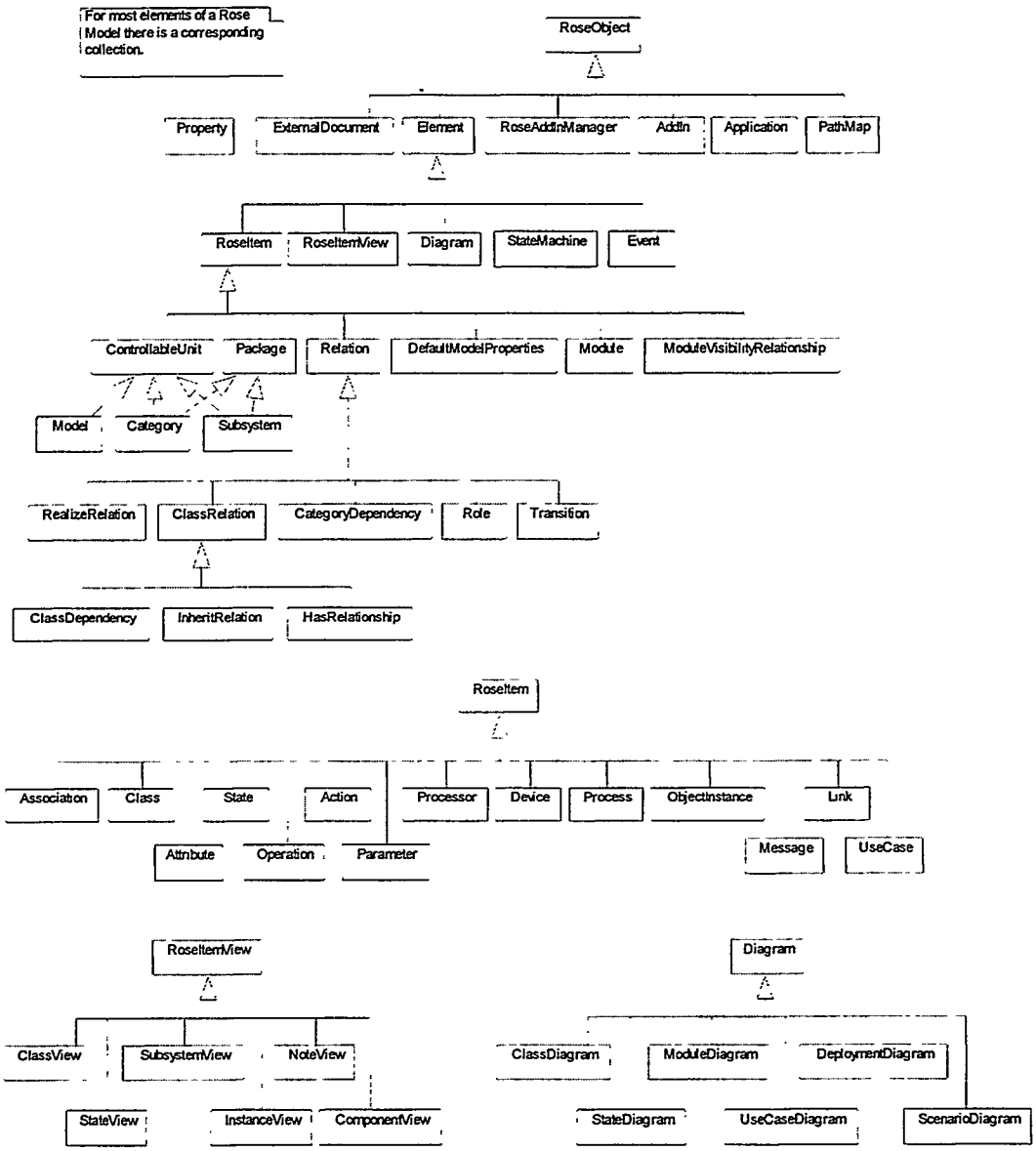


Figure 7-5. Rational Rose 98 REI Class Diagram.

A file called a type library makes the REI available for use in the automation environment. The type library provides a standard description of the interfaces that make up the REI. The type library is used with a tool named *JavaTLB* to create a set of Java *wrapper classes* for the interfaces of the REI[3] (JavaTLB is provided with Microsoft Visual J++ 1.1, Java Development Environment). The REI interfaces can then be accessed from Java directly.

The classes in the Rose API interface category provide an intermediate interface between the other OPRRA classes and the Java wrapper classes. Other OPRRA application classes that need information from Rose use these classes. The classes hide the details of working with the REI objects and provide a layer of abstraction between OPRRA and Rose. Any changes to Rose in the future should affect only these classes. A tool different from Rose could be used with OPRRA by adapting these classes to the new application.

Figure 7-6 shows the class diagram for the OPRRA application. The classes that make up the Rose API interface category are RApplication, RModel, RClass, RState, RAttribute, and RTransition. These classes directly relate to the corresponding REI Java wrapper classes IRoseApplication, IRoseModel, IRoseClass, IRoseState, IRoseAttribute, and IRoseTransition. The interface classes duplicate some of the information in the REI classes and add functionality. For instance, in Rose the user can define a transition that takes an action external to its class by referring to a transition defined in the other class. Within Rose, the external transition information is entered as character strings. The RTransition class maintains this reference but translates the strings into the corresponding RTransition. For another example, as part of the methodology, a system object is entered as a class, assigned an attribute, and a *type* class in the model defines the type of the attribute. The attributes of the type class are the valid values of the referring attribute. The RAttribute class understands this relationship and also keeps the valid values for the attribute in the class.

OPRRA is used with a single running application of Rose. Consequently, OPPRA creates a single instance of the RApplication class. In turn RApplication contains a single instance of RModel. However, there may be a number of class, state, attribute, and transition objects defined in the Rose model. There will be a corresponding interface class instance created in OPRRA for each of these objects. The interface classes are somewhat centered on RState. The RState class contains a reference to its parent class as an RClass. The RClass contains the attributes defined for the class as RAttributes. The RState also maintains collections of the transitions that are coming into the state and those leaving the state as RTransitions. The ProcessModel class of OPRRA keeps a reference to the RState that the process model belongs to.

Rose is used to model a system by including a state model for each class. The state model defines a state machine that shows the states and the transitions between states for the class. The user starts the OPRRA application and selects a state from the Rose model. The user selects the OPRRA menu item *Browse, Process Model.* When this happens, the OPRRA class uses RApplication to determine which state is selected on the

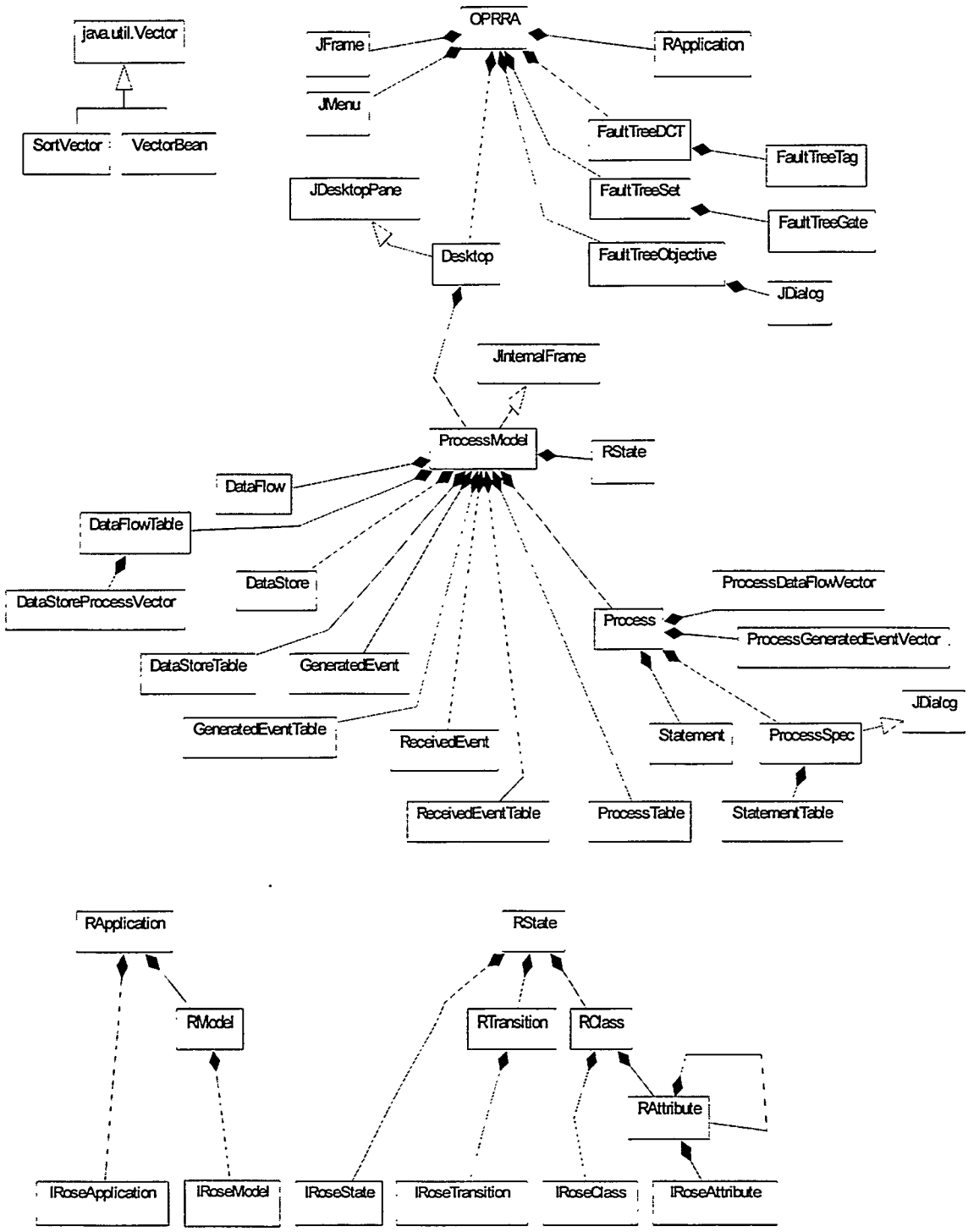


Figure 7-6. OPRRA Application Class Diagram.

state diagram in Rose. An instance of ProcessModel is created, with a new RState class created for the selected state. When the new RState is created, data for the corresponding state in the Rose model are retrieved and other interface classes related to RState are also created. For instance, the state in Rose belongs to a class. Consequently, an RClass is created corresponding to that class and a reference to the RClass is kept in RState. Also, the state in Rose may have transitions associated with it. In this case, new RTransitions are created, corresponding to each transition, and references to these are kept in the RState. When the RClass is created, it finds the attributes in Rose for the corresponding class and creates new RAttributes, one for each of the attributes. In this way the classes in the Rose model are mirrored in the interface classes of OPRRA.

7.3.3 Process Model Category

The majority of the classes for the OPRRA application are used to capture the process model for states in the Rose model. The process model is defined by a DFD. The five main elements of a DFD are process, data store, data flow, received event, and generated event. A process contains a collection of statements that define the procedure for a process.

There are corresponding classes in this category for each of the DFD elements, which are named Process, DataStore, DataFlow, ReceivedEvent, GeneratedEvent, and Statement. As the user creates new instances of the process model elements, new instances of these classes are created and stored in collections. The ProcessModel class contains collections for Processes, DataStores, DataFlows, ReceivedEvents, and GeneratedEvents. The Process class contains a collection for statements.

There are also classes to display the contents of the collections for each of the DFD elements. These are named ProcessTable, DataStoreTable, DataFlowTable, ReceivedEventTable, GeneratedEventTable, and StatementTable. The collections are displayed in a table format. Each column of the table corresponds to an attribute of the class and each row of the table corresponds to one of the classes in the collection. For instance, a DataFlow has three attributes; the source of the DataFlow, the attribute that is *carried* by the DataFlow, and the destination of the DataFlow. Consequently, the table has a column for source, attribute, and destination. If the DataFlow collection in the ProcessModel contains three DataFlows, the table will have three rows, each one displaying the data contents of the corresponding DataFlow.

The ProcessModel contains the ProcessTable, DataStoreTable, DataFlowTable, ReceivedEventTable, and GeneratedEventTable. The ProcessModel class extends (inherits) an internal frame GUI component. The tables for the ProcessModel are displayed in tabbed panes contained by the internal frame. The ProcessSpec class contains the StatementTable. ProcessSpec is a modal dialog box that is *Popped up* by process to allow a user to enter statement data.

Several other classes are included in this category to process information. The DataStoreProcessVector is a class that is a collection of all DataStores and Processes that are defined for the ProcessModel. This is used by the DataFlowTable to display a list of

valid sources and destinations for a DataFlow. The ProcessDataFlowVector is a class that is a collection of DataFlows that have a specified process as either the source or destination. This is used by Process to create two collections of DataFlows, one for the DataFlows that are coming into the process and one for those that are leaving the process. The ProcessGeneratedEventVector is similar in that it is a class that is a collection of GeneratedEvents that have a specified process as the source of the GeneratedEvent. This is also used by Process to create a collection of GeneratedEvents that are leaving the process.

7.3.4 Fault Tree Support Category

There are five classes that support the generation of a fault tree, which are named FaultTreeDCT, FaultTreeSet, FaultTreeTag, FaultTreeGate, and FaultTreeObjective. In addition, there are methods in the Desktop, ProcessModel, and Process classes that are used to generate the parts of the fault tree for data contained in these classes.

The fault tree data can be rendered as two separate files to make it compatible with the ArrTree application. The two files are the *Set* file (.set extension) and the *DCT* file (.dct extension). The Set file contains a list of *gates* that describe the fault tree. The gates are logic gates such as AND and OR gates as well as special gates defined for risk and reliability analysis. A gate contains references to the other gates that are an input to the gate or an output of the gate. The gate also contains a descriptor. The descriptor for the gate varies with the gate's location in the fault tree and is stored in the DCT file. For instance, a gate that is created for a state will specify the state and one created for a process will specify the process.

The FaultTreeGate class encapsulates the information for a gate. A FaultTreeGate is created by specifying the type of the gate and the descriptor for the gate. Input and output gates can be added to the gate using methods provided by the class. The gate descriptor is specified in terms of the FaultTreeTag class. Methods that generate fault tree data (e.g. in Desktop) create new instances of FaultTreeTag and FaultTreeGate as needed. The new instances are added to a collection of the classes kept by the FaultTreeDCT and FaultTreeSet classes respectively. When the fault tree is complete, methods are called to FaultTreeDCT and FaultTreeSet to write the data stored in the class to their respective files.

7.4 Software Availability

Interested persons within Sandia National Laboratories can examine the software described in this chapter directly from Sandia's Internal Restricted Network. Rational Rose 98 and OPRRA applications are available on a server named *FAAServer3* in the Sandia Domain. The subdirectories containing the applications are shared on the server; however, you must be on the access list for the subdirectories before you can use them. Rational Rose 98 is in the *Rose98Ent* subdirectory and OPRRA is in the *OPRRA11* subdirectory. Both of these subdirectories contain a subdirectory named *aaaSetup Info* that has a Word document providing installation instructions. To use the applications, the

two subdirectories are connected to your PC as network drives. The installation instructions create registry entries on your machine to run the applications from the network.

In the *OPRRA11\samples* subdirectory, you will find a Rose model file, an OPRRA serialization file, and Set and DCT files containing a fault tree generated from the corresponding model data. In the *OPRRA11\doc\api* subdirectory is the documentation for the classes used in the OPRRA application. These files are HTML files that are generated by the *Javadoc* tool directly from the source code (Javadoc comes with the Sun Java Development Kit). The source code for the OPRRA classes can be found in the file named *src.zip* in *OPRRA11*. The file must be unzipped to view the source code. The source code is placed in a subdirectory named *sandia\oprra* under the directory from which the file is unzipped.

Users on the Sandia National Laboratories Internal Restricted Network can also download the risk and reliability analysis tool used in this chapter. ArrTree is available by itself or as part of the SABLE/ARRAMIS[2,4] package from the URL:

http://www.csu821.sandia.gov/organization/div6000/ctr6400/pra6412/

Users outside of Sandia National Laboratories can obtain ArrTree, SABLE, and ARRAMIS software under a licensing agreement that can be arranged through Sandia's Partnering and Licensing Department. Note that it is a violation of copyright agreements as well as both Department of Energy and Sandia intellectual property policies to provide this software to persons outside of Sandia National Laboratories without obtaining a signed license agreement *first*.

7.5 Interfacing with Commercial Surety Analysis Software

There are currently many software packages available to perform varying portions of the surety analysis problem from both commercial and noncommercial entities such as Sandia National Laboratories. It is neither desirable nor efficient to try to recreate the capabilities of those codes within the OPRRA application. Rather, it is our desire to construct a framework under which an intermediate layer of software could extract the surety analysis models required by those other software products. This is the approach that has already been used in the area of fault tree analysis: a Java interface was constructed to extract fault tree logic from the common object model and translate it into a format used by the SABLE fault tree analysis software (found in the ARRAMIS risk and reliability analysis workstation). This section notes some of the requirements that will likely have to be imposed in order to make the cooperation between OPRRA and such other software products both possible and convenient.

7.5.1 Characteristics of Surety Information Extraction Software

Since the OPRRA application framework represents the embodiment of new research concepts (a framework for using object-oriented analysis methodology to construct a variety of risk and reliability analysis models), there is likely to be no way for an existing surety analysis software product to interface directly with OPRRA. In addition, since the traditional surety analysis models are not explicit, but are encapsulated within the behavioral models developed in OPRRA, it is not possible to have current versions of these traditional surety analysis products directly translate the OPRRA database into an appropriate surety model. For this reason, we believe that the most efficient way to gain utilization of the methods found in OPRRA is to build small software modules that extract traditional surety analysis models from the OPRRA database and translate those models into the format used by each particular traditional surety analysis software product. These small software modules will not be as simple as, say, a file format translator because they must embody a rule set that will construct appropriate queries for the OPRRA database, interpret the results of those queries, and then translate those results into the appropriate model and file format. The rule set in particular may be non-trivial. Appendix A contains an example rule set for the extraction of fault trees from the OPRRA database. The rules for extracting other types of surety models will likely be at least as complex as those found in the appendix.

The first such software module to embody such rules extracted fault trees from the OPRRA database for analysis by the SABLE software. The characteristics of this module are described elsewhere in this chapter. However, the general characteristics of this module will likely serve as a model for other such modules. These characteristics include operation in the Microsoft Windows NT environment, the ability to specify the goals for which a particular surety model is to be generated, the embodiment of the rules required for generating the model, and the ability to make the appropriate application program interface calls in order to access the OPRRA database as managed by the Rational Rose object-oriented analysis software. While it is certainly possible to build the module as a separate application (a separate Java application was developed for extraction of the SABLE fault tree model), it would be highly desirable if this module were built into the surety analysis software product because it would eliminate potentially confusing steps for the user/surety analyst. If the module were incorporated into the surety analysis product, the analyst would simply open the surety analysis product and specify the goal for which a surety analysis was being sought, as well as the Rational Rose object model database that contained the appropriate common object model. Once this was specified, the module could be invoked to automatically extract the model, and the model could be inserted directly into the software's internal database. The software could also query the user for any special cases or unresolved conditions relative to the requested analysis, if necessary. It would then be a seamless transition from model importation to model solution and the expression of surety results.

7.5.2 Surety Analysis Software Requirements

Section 7-1described a method for enabling existing surety analysis software to make use of the common object model paradigm as implemented in the OPRRA demonstration

software. The long-range goal would be to enable the integration of currently disparate surety analysis software into a single coupled entity that would behave as an integrated surety analysis workstation. In order for such a vision to be realized, it would require significant modifications to the existing surety analysis tools to make them compatible with the OPRRA framework. In the short term, while OPRRA is operating in a strictly PC/Microsoft Windows NT environment, this would mean converting the existing surety analysis software into callable modules (for example, dynamic link libraries or OLE objects) that could be called from under an integrated user interface. In the long run, should OPRRA be modified to run as a network-based client/server application, it would be more beneficial to convert individual tools into CORBA objects that could be accessed through a network environment. The incremental benefits of such an integrated environment over a well-integrated but stand-alone tool (as described in the previous section) may not be large enough to justify conversion for the strictly PC/Microsoft Windows NT environment unless a significant core of regular users has been assembled. The exact tasks to be accomplished to make such a conversion would be highly specific to the individual surety analysis tool, so further description of this process is beyond the scope of this report.

7.6 Realizing the Vision of Integrated Surety Analysis

The current OPRRA application should be considered a *proof of concept*. It demonstrates some of the basic functionality, but is by no means complete. However, it is felt that the basic approach is sound and can be built upon.

7.6.1 Improvements

More work needs to be done to make sure that Rational Rose is a good foundation for modeling a physical system in ways that are desired for risk and reliability analysis. Some of the ideas that were encountered as the methodology developed did not seem to be well suited for a tool that is primarily used for software development. This may warrant replacement of Rose with a more suitable tool, or the development of a custom tool to replace Rose. This will have some impact on OPRRA but it is hoped that many of the classes can be reused.

The fault tree generation needs more testing and refinement to make it more usable. We discussed and did some preliminary work on implementing other data generators for methods such as event trees and FMEA. It was felt that these methods were reasonable to implement; however, there was not enough time or resources to do these.

7.6.2 Client-Server Architecture

If usefulness and user support warrant, the tool could be evolved to a distributed type of application that would allow multiple users to more easily collaborate on analysis. This would involve moving to a *client-server* architecture. This might be done in a way that would support Unix workstations as well as PCs.

A database should be used to organize data and provide positive control of the data among multiple users. The database would run on the server and store modeling data as well as analysis results. Since Rose and OPRRA are designed using OO, methods it would be best to use an object-oriented database (OODB). Earlier in the project we did some prototype development work using the Versant OODB. Versant provides an easy way to convert existing classes so that the data of the class are persistent and available in the database.

A way to communicate between client and server may be needed. Given the OO nature of the application, the best technical solution is to use CORBA. CORBA provides a way for classes to interact across a network without regard for which machine the class is actually running on. CORBA would be used as needed where a class on the client needs to cooperate with a class on the server to perform some function. Databases such a Versant already provide a client-server architecture, so it may be possible to create a solution where CORBA is not needed.

The first major problem would be how to adapt Rational Rose. In fact, this might be a big enough problem that it would warrant using a different modeling tool. Rose is not designed as a distributed application and does not use a database. Data are maintained in the application and saved to a file in a custom format. One could probably write an application that would pull all of the data out of Rose via the REI and store it in corresponding objects in the database. To load a saved model, the application would have to perform the reverse process, using REI to set all of the classes in Rose to the saved data from the database. Another way might be to interpret the Rose model file to create the objects in the database. This would require knowledge of how the Rose model file is formatted, which may be proprietary. The model file could be saved in the database as well, so that it can be accessed when loading a saved model. With this method, only the subset of the Rose data that is really needed would be interpreted and saved in the database.

Rose would have to be installed on each client machine. This can be simplified by providing an application server on which software actually resides and that the client attaches to by using a shared drive.

Since Rose must be installed, OPRRA might as well be left as an application. If circumstances were different, it would be quite easy to convert OPRRA to an applet that could be loaded from a web server. An attempt was made to separate the classes in OPRRA that store data from the classes that display the data. It should be fairly straightforward to modify the data classes to put them in the database. Some additional work may be needed on OPRRA to better separate the data classes from the display classes.

Results of analysis, such as fault trees, should be stored in the database in the corresponding classes. The data can then be rendered as needed for use by other applications. For instance, ArrTree requires two files in a specific format. Another application might need the data in some other format that could be files or other objects

in the database. This would be handled by adding new methods to the fault tree classes to generate the desired format from existing data.

Existing and new applications would need to interface to the database to work with the Rose and OPRRA data. They might interface directly to the database, or an intermediate application could be written that interfaces to the database and translates data into a format that is compatible with the application.

7.6.3 Interface to System Design Tools

The final – and most long-term – vision for the products of this project would involve the integration of the successors to the surety analysis tools described in this section with current- or next-generation computer-aided system design tools. While the reasons for this proposed merger of tools are clear, the task of performing such integration would be immense. For that reason, this section will focus on the vision for the integrated product and provide only a limited description of the tasks that may be required to achieve it.

The current method for performing surety analyses has been referred to by some as the "volleyball method." In this method, the design team develops a candidate design in isolation from the surety analysis team. This candidate design is then "tossed over the net" to the surety analysis team, who makes assumptions and analyzes the candidate design. These recommendations are then once again tossed over the net for consideration by the design team. During this time, however, the design team has been assessing the candidate design and making their own revisions (in the absence of input from the surety team). Thus, it is not unusual for the design team to already have recognized and corrected some of the flaws in the candidate design before they receive input from the surety team. Furthermore, each time the candidate design is revised by the design team, the surety analysis team must discover for themselves the differences between the new and old candidate designs, and then assess the relevance of these design changes to the surety issues. As a result, the surety analysis team is rarely assessing the most current design – they are almost always at least one generation out of date. This is often a cause for tension between the design and analysis teams, and it is obviously a source of considerable inefficiency. The key problems are caused by the fact that a human must examine each new design and translate that design into the relevant surety models. The constant repetition of this step is costly and inefficient, and we believe unnecessary, given an appropriate design and analysis environment.

The ultimate vision for our surety analysis system would be to eliminate this volleyball approach and replace it with a system in which the surety analysis tools could look directly into the databases generated by the computer-aided system design tools and abstract from them a reasonable common object model for surety analysis. In such a system, the surety analysis tool would be able to look into the design database and identify the types of components that are being used as well as the logical and physical interconnections between them. The design tool may also contain information about the boundary conditions of the system (expected and extreme operating conditions, functional requirements and specifications, etc.) that could be directly translated into logical statements for which surety analyses could be performed. The surety tool would

draw entries from its predefined library of generic objects that represent the particular components found in the design database and connect them according to the physical and logical connections seen therein. The analyst would then select the types of surety analyses to be performed on this new model (most likely focusing on areas that have changed since the previous iteration). As the results of these analyses are generated and validated, they could quickly be fed back to the design team. In this way, the surety analyst would always be assessing the most current candidate system design and be able to provide feedback to the design team in a timely manner, potentially reducing or eliminating many of the design/redesign cycles experienced by current design engineers.

The convergence of surety analysis tools and system design tools as envisioned here would be an extraordinarily ambitious undertaking. It would require negotiations between surety and design engineers (as well as their software developers) to agree on not only the format of such a design database, but even on such issues as the data to be contained in it, which data can be modified by designers versus surety analysts, ownership and quality assurance of design and surety data, whether designers would be required to enter large amounts of data that are not currently required, and so forth. It would require the development of new levels of trust between these two disciplines, which have at times experienced hostile relations in the past (with designers viewing surety analysts as independent auditors or enforcers).

Thus the achievement of this vision requires not only huge technical advances, but also political will and possibly cultural changes. As such, it is not possible to predict the exact sequence of tasks or steps that would be required to achieve its ultimate success with any certainty. However, we believe that it is a logical ultimate goal given the inefficiencies, errors, and communication problems that plague the current design and surety analysis paradigm. As surety analysis techniques become more common in industrial applications, design and analysis software makers may conclude that there would be a significant market for such a product if it were to be introduced.

7.7 Summary

A software application named OPRRA has been developed to demonstrate the fundamentals of a methodology to apply object-oriented analysis to systems for risk and reliability analysis. It provides a capability to enter action data flow diagrams that model the process for object states and to generate fault trees from an object-oriented system model. Improvements are needed and enhancements would be useful to make the application more capable; however, the application can be used as the basis for further development.

7.8 References

1. Shlaer, S. and Mellor, S., *Object Lifecycles: Modeling the World in States*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

2. Wyss, G.D., A Training Course for the ARRAMIS™ Risk and Reliability Analysis Software Suite, Sandia National Laboratories, Albuquerque, NM, to be published.

3. Orfali, R., and Harkey, D., *Client/Server Programming with Java and CORBA*, Wiley New York, 1997.

4. Hays, K.M., Wyss, G.D., and Daniel, S.L., A User's Guide to SABLE 2.0: The Sandia Automated Boolean Logic Evaluation Software, SAND96-0842, Sandia National Laboratories, Albuquerque, NM, 1996.

8 Limitations of the Method

During the course of this research we discovered a number of potential limitations for this method. Some of these appear to be inherent in the object-oriented methodology and/or the risk assessment methodologies that it is intended to support. Others may be resolved through additional research.

8.1 Inherent Limitations

The risk and reliability analysis methods that are used to support the object-oriented analysis methodology are inherently limited to systems that can be described in terms of discrete states. Thus systems that can only be described in terms of a continuous modeling space will be difficult to represent in a meaningful manner using this methodology. Granted, one could go to finer and finer discretization of the continuous space to simulate the use of continuous variables, but this would cause the discrete modeling space to become extremely large and potentially intractable. Since one of the fundamental assumptions of the underlying risk and reliability analysis methods (especially fault tree and event tree analysis) is that of the discrete-event space, we believe that other analysis methods will likely provide better results for systems that defy discretization.

A second limitation that may also be inherent to the methodology is related to the concept of nondeterminism. A significant number of systems — especially multitasking computer systems — exhibit nondeterministic behavior. The nondeterminism stems from the fact that a particular process may exhibit different delay characteristics, depending upon what other processes are executing on the computer at the same time and how the computer's resources are divided among those processes. Thus one cannot always be sure that two processes will be completed in the same order. This is particularly true for interrupt-driven systems. Some object-oriented analysis methodologies contain elements that allow one to consider such nondeterminism. However, nondeterminism is difficult to map in the space of risk assessment models. Some nondeterminism can be incorporated into the model through the use of random events. These events can then be incorporated into fault tree and event tree analysis models. But, in general, the qualitative reasoning concepts that are required in order to extract risk analysis models require that a large degree of determinism be present. Thus systems that are highly nondeterministic will be very difficult to model using the methodologies described in this report. It should be noted, however, that the subject of nondeterminism is a very active research area that is by no means completely resolved — even when the requirements of the risk analysis methodologies are not present.

8.2 Recommended Future Research

The methods described in this report were demonstrated to apply to a large variety of systems and risk analysis methods. However, owing to the constraints of time, it was not possible to demonstrate that one could derive all possible risk analysis methodologies

from the object-oriented model. A reasonable next step for research would be to take other commonly used risk and reliability analysis methodologies and determine a rule base by which those models could be derived from an object model such as the one described in this report. While it is believed that these methods can be supported with relatively minor additions to the object-oriented modeling methodology, that fact remains to be demonstrated through further research.

The constraints of time also limited this project largely to two of the five "views of the systems" that are required to appropriately understand all aspects of the system (the functional and physical views). While consideration was given to the environmental, temporal, and life cycle views, they did not receive the attention that would be required to claim that they have been fully demonstrated or implemented in this method. Further research to understand the mappings and special details that are required for those three additional views would help complete this work.

The concepts of qualitative reasoning were embodied in the object methodology through the use of state transition diagrams, data flow diagrams, and truth tables. This implementation is adequate for many of the risk and reliability analysis models that were postulated. It is believed, however, that these constructs may not be appropriate to model some classes of systems – particularly those that may be characterized by more continuous variables (as described in Section 8.1). The theory of qualitative differential equations may provide valuable insights into these systems and may eventually be found compatible with the fault tree and event tree methodologies that we have described in this report.[1]

8.3 References

1. Kuipers, B., *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*, MIT Press, Cambridge, MA, 1994.

9 Summary and Conclusions

9.1 Conclusions

This report documents the work conducted by a multidisciplinary internal research project to create an extensible framework capable of supporting a broad range of surety assessment techniques based on concepts from object-oriented analysis. As a result of this work we concluded several things:

- There is a great deal of similarity in the information required by system designers, simulators, and surety analysts. However, under current practice, each discipline – indeed each technological domain – stores this same information in very different forms.

- Concepts derived from object-oriented analysis can be used to form an excellent common language (and hence the basis for a common knowledge repository) for the information required by all of these disciplines.

- Models required to perform surety analyses (especially risk and reliability analysis models) can be derived from the object-oriented analysis methodology provided that the concepts embodied in the object models support both inductive and deductive qualitative reasoning. The surety analysis models that are derived from such common object models are self-consistent.

- Object models that embody state transition diagrams, events, process flow diagrams (similar to data flow diagrams), and two-way qualitative transfer functions (such as our truth tables) are adequate for the derivation of many surety analysis models, including failure modes and effects analysis, HAZOP, event trees, and fault trees. These surety models can be derived from an appropriate object model using an automated query engine that embodies relatively simple logical rules.

- When the UML language is extended to incorporate data flow diagrams, the major elements of this modeling methodology will be representable in a standardized language for which commercial off-the-shelf software is already available.

- Object models can embody not only the physical composition of a system but also its interactions with its environment and its functionality (its "purpose for existence"). This can lead a surety analyst directly to an assessment of the consequences of a system behavior in terms of the states in attributes found within a modeled scenario. This can automate the ranking of scenarios developed in the failure modes and effects analysis, HAZOP, and event tree modeling methodologies.

- While this research did not directly address issues of the temporal and life cycle views of a system, a conceptual methodology was developed to map those issues back

into the object model. The details of that mapping would be the subject of future research.

- More advanced qualitative reasoning engines, such as qualitative differential equations, may provide the basis for even better surety analysis models and reasonable simulation tools based on the object modeling methodology.

- Because of the requirement that this modeling methodology support both inductive and deductive reasoning, it is unlikely that it will adequately support the analysis of nondeterministic systems. It should be noted, however, that most object-oriented modeling methodologies – and indeed most risk and reliability analysis methodologies – also produce inadequate results for nondeterministic systems.

The most important result from this study is the conclusion that it is reasonable to work toward a repositioning of common knowledge that can be used by system designers, simulators, and surety analysts which embodies, in abstract form, the detailed behavior of a system. Such a knowledge repository would not only aid communication among different engineering disciplines but would also relieve the surety analyst of the need to interpret and recreate much of the design information for incorporation into the risk and reliability analysis models. The fact that such models can be derived directly and automatically from an appropriately constructed common knowledge repository will enable the surety analyst to spend much more time assessing the strengths and weaknesses of a system, and correspondingly less time in the nonproductive details of constructing the syntax of his models. In other words, the surety analyst will be enabled to ask far more "What if?" questions about a system in a shorter time than ever before. This will result in either a more detailed analysis of the system, a faster turnaround time for the surety analysis, or both. It should also enable surety analysis techniques to be performed on new systems for which they would have previously been prohibitively costly, resulting in safety improvements for a wide variety of products.

9.2 Summary

The purpose of this report has been to introduce the reader to the ways to use a common object model to perform a surety analysis of a "real" system. Such a surety analysis uses techniques derived from traditional risk and reliability analysis methods. It is based on the premise that an analyst makes explicit his knowledge of the performance and behavior of a system in an appropriate object-oriented model. Such a model can draw from a knowledge base of existing, similar objects so that the analyst can incorporate the known failure modes and vulnerabilities of those objects with minimal effort. Since the model supports both inductive and deductive qualitative reasoning, it is possible to automatically derive many types of surety analysis models based on a single common object model. The automatic creation of these surety models enables a system analyst to perform a greater number of analyses of a more consistent quality than would ever be possible under the current "hand-tooled" model development methods. It also provides a way to improve communication among system designers, simulators, and surety analysts. While the ultimate development of a comprehensive tool to embody this method in a

production environment may be some time away, the concepts proved under this project form a logical basis for a major and beneficial change in basic engineering design and analysis methodology.

Appendix A Rules for Extracting Logic Models from the Common Object Model

A.1 Fault Tree Development Rules

Note: This set of rules is applicable for those properties that are expressed as a series of mutually exclusive discrete attributes for which order *cannot* be expressed. It will also put out a *reasonable* result for those properties that represent discretizations of continuous ranges (where one can make statements such as "less than" or "greater than" a value). However, additional modifications to these rules would be desirable in order to produce more optimal trees that use those criteria implied by ordering of the discrete values for those parameters.

1. Select a "normal" state for the system model.
   a. Which state is normal
   b. Which attribute values are normal

2. Define the valid states and conditions.
   a. Valid versus invalid conditions
      i. List any states that are impossible or out of bounds as initial conditions.
      ii. List any combinations of discrete attribute values (DAVs) that are impossible or out of bounds as initial conditions.
   b. Consider both initial conditions and operational conditions.

3. Define the objective.
   a. Develop a verbal definition of the objective.
      i. If the objective is stated negatively (i.e., these conditions must not occur), it should *not* be embodied in the "normal" state described in (1)
      ii. If the objective is stated positively (i.e., ensure that these conditions occur), it should include, but not necessarily be limited to, the "normal" state
   b. Translate the verbal objective definition into a logical statement in terms of DAVs and/or states.

c. If necessary, expand and minimize this logical statement to obtain "objective cut sets." Each objective cut set represents one unique combination of conditions that must either be avoided (for negative objectives) or ensured (for positive objectives).

d. Examine the objective cut sets and remove any that are physically inconsistent (e.g., a cut set that requires the system to be in two incompatible states at the same time).

4. Develop the top event of the fault tree

   a. The top event is an OR gate.

   b. Inputs to this gate consist of one or more OR gates — one for each objective cut set developed above.

   c. In addition to this, one must also deal with the "normal" state.

      i. For positive objectives, this should already be included in the objective cut sets because the objective should include the normal state.

      ii. For negative objectives, one must place an additional input to the top event: an OR gate that represents "failure to achieve the normal operating state."

5. Examine object model for instances of the objective cut sets.

   a. If an objective cut set contains a state as one of its constituent elements, look only in that state to see if the requisite DAVs can occur (as results of truth tables or as inputs that are external to that object).

      i. If it cannot occur, place the FALSE basic event as the only input to the OR gate that represents this objective cut set.

      ii. If it can occur within that state, then *each* condition under which it can occur (as defined by the truth tables for that state) must be represented by an AND gate that forms an input to this objective cut set's OR gate.

   b. If an objective cut set does *not* contain a state, look at *all* states within the object to see if the requisite DAVs can occur (as results of truth tables or as inputs that are external to that object).

      i. If it cannot occur in *any* state within the object, place the FALSE basic event as the only input to the OR gate for this objective cut set.

      ii. Otherwise, make a list of all states within the object where these conditions can occur. Each such state becomes its own OR gate that is an input to the objective cut set's OR gate.

      iii. Furthermore, *each* condition under which these conditions can occur (as defined by the truth tables for that state) must be represented by an AND gate that forms an input to this objective cut set's OR gate.

6. Normal branch of the tree (for negative objectives — assuming only 1 normal state)
   a. Examine the normal state and from the truth tables for that state, list those DAVs that come from outside of the object that are required in order to attain the normal state.
      i. Each such DAV is represented as an OR gate that is an input to the "failure to achieve the normal operating state" OR gate.
      ii. The OR gate represents "failure to attain the correct DAV in order to obtain the normal state." Its inputs are all DAVs for that attribute *other than* the one that leads to the normal state.*
   b. Search all states in the object to determine any events that cause the object to transition into the state associated with "normal."
      i. Each pair (an event that can cause transition to normal, a state that is being transitioned from) represents an AND gate that is an input to the "failure to achieve the normal operating state" OR gate.
      ii. The AND gate represents "failure to transition from this state to the normal state." Its inputs are:
         (1) Object is in this state, AND
         (2) Event to cause transition does not occur

7. Expand the fault tree: The following process is applied recursively until stopping criteria are met for all "leaf nodes" of the tree.
   a. Within a particular state, each event or DAV that is represented by a gate input within the fault tree is resolved into its immediate causes.
      i. If internally generated, its causes are found by examination of the truth table from which it comes. The truth table itself is represented by an OR gate; each line in the truth table that produces this DAV is represented by an AND gate with inputs that are the entries on the input portion of that line in the table.
      ii. If externally generated, its source(s) are noted (a DFD from another state in this object; another object; and/or external to the system). The event or DAV itself becomes an OR gate and each possible source represents an input to this gate. Resolving this event requires tracking through state and object transitions.
      iii. If the DAV or event can be generated *both* internally and externally, follow a combination of (i) and (ii) above. The event or DAV itself becomes an OR gate, and each possible source represents an input to this gate. The internal truth table is

---

* Note that this assumes only one normal configuration. Extensions to this rule are required if multiple combinations of DAVs can lead to the normal state.

an additional input to this OR gate and is itself an OR gate as described in (i) above. The remainder of (i) is then carried out to resolve the truth table.

b. If a state itself is a direct input to a gate within the fault tree, the way that the object can enter that state must be resolved into its immediate causes.

   i. Search all states in the object to determine the sources of any events that cause the object to transition into the named state.

   ii. Search all external entities (e.g., other objects) to determine the sources of any events that cause the object to transition into the named state.

   iii. Each pair (an event that can cause transition to the state, and a state that is being transitioned from) represents an AND gate that is an input to the state's gate. The AND gate represents "transition from the previous state to this state." Its inputs are:

      (1) Object is in previous state, AND

      (2) Event to cause transition occurs

c. If an input to a gate within the fault tree is an event or DAV that can be generated by another object, all of the above rules apply.

d. As each fault tree gate is constructed, the algorithm should seek to determine whether the inputs to that gate would require the system to be in an invalid state (e.g., a physically impossible combination of DAVs and/or states and/or events would be required). If such a situation can be identified, then all of the inputs to this gate can be removed and replaced by a single FALSE event.*

8. Stop the recursive fault tree expansion process whenever any of the following criteria are met:

   a. Stop if the event or DAV is generated outside of the system being analyzed (place that external event in the fault tree and stop further development).

   b. Stop if adding the next event or DAV will cause a logical loop (i.e., if one can travel directly up toward the top of the fault tree and encounter that *same* logical event).

      i. Place a "developed event" there for the loop behavior

      ii. Warn the analyst**

---

* Note that it may be difficult identify such situations a priori because portions of the impossible combination may appear in another branch of the fault tree. That is why this method suggests that the cut sets be examined for physical consistency before being accepted and quantified.

** Rules for dealing with logical loops will be developed at a later time

c. For a negative objective, stop if the logical condition found (i.e., truth table line) is identified as part of the "normal" state of operations (recall that "failure to achieve normal status" was handled explicitly at the top of the tree).

   i. Place a developed event in the fault tree to represent this situation, with its logical value set to TRUE.

   ii. Provide the analyst the option to continue expansion beyond this point if desired.

d. For a positive objective, stop if the logical condition found (i.e., truth table line) is *not* part of the "normal" state of operations

   i. Place a developed event in the fault tree to represent this situation, with its logical value set to FALSE.

   ii. Provide the analyst the option to continue expansion beyond this point if desired.

9. Fault Tree Solution: The fault tree is solved using traditional tools such as SABLE. If quantification and/or quantitative truncation of the cut sets is to be performed, then the quantitative data must be exported (in addition to the fault tree itself) in an appropriate format for use by the solver.

10. Cut Set Examination

    a. Until we have extensive validation experience on the fault trees and cut sets produced by this modeling technique, each cut set should be examined to ensure that it is in fact (1) an actual failure mode of the system, and (2) achievable without violating consistency requirements imposed by the object model.

    b. The cut sets can be ranked quantitatively and/or qualitatively to provide insights to the analyst.

       i. Qualitative: rank by number of basic events in each cut set

       ii. Quantitatively: rank by relative probability or frequency of the cut sets

       iii. Consequences: rank by the relative consequences that would be expected to occur if the cut set were to occur

       iv. Risk: traditionally, risk is represented by frequency times consequences, although modern studies may represent risk as a nonlinear function of frequency and consequences that is appropriate to the situation at hand.

    c. The cut sets can be processed quantitatively to determine the relative importance of each basic event to the overall consequences, risk, etc., using the traditional importance measures (risk increase, risk reduction, partial derivative, Fussel-Vessley, etc.).

    d. All of the above rankings and importance computations can be performed as part of an uncertainty analysis in which the probabilities and other parameters are sampled using Monte Carlo techniques to obtain statistical distributions for each measure (except, obviously, the qualitative ranking).

A.2 Event Tree Development Rules

1. Specify an initial state for the system model.
   a. State of each object in the model
   b. All discrete attribute values

2. Define any invalid states and/or conditions.
   a. List any states that are impossible or out of bounds.
   b. List any combinations of DAVs that are impossible or out of bounds.

3. Select one of the following for an initiating event for the event tree.*
   a. The system is in a normal steady state.
   b. An event occurs.
      i. External influence on the system
      ii. An event is generated within the system — either by a "known cause" or simply a "postulated event" (i.e., unknown or unspecified cause)
   c. One or more DAVs change
      i. External influence on the system
      ii. An event is generated within the system — either by a "known cause" or simply a "postulated event" (i.e., unknown or unspecified cause)

4. Select a starting point for the event tree model.
   a. If starting from a steady state, ask the analyst which object is to be used for the starting point of the event tree. The analyst must realize that any objects that are solely upstream of this point in the model will not be included in the event tree model.
   b. Otherwise, the event tree begins with the object where the event or changed DAVs first enter the system.

---

* If the event or DAV changes affect more than one object in the model, then additional rules must be developed to determine how the event tree begins and progresses. This is a subject for later development.

5. Propagate the conditions through the object model to develop an event tree.
   a. Method 1: Random events only
      i. Examine the initial object model state for the object identified as the event tree model's starting point. If any random events can affect this object (the DFD for the initial state of this object), split* at that point.
      ii. Apply the initiating event to the object model. Propagate the flow of the system into new states as appropriate. As each new state is entered, examine its DFD to determine if any random events can affect its flow, and if so, split at that point.
      iii. As each new object is entered, apply the above two steps to that object and split as necessary. Examine the path to see whether it remains within the restrictions specified in the list of valid states and conditions. If the path is found to be invalid, truncate it.
      iv. The model cannot form a logical loop by going through the same object *and* state more than once.** Consider the case where the model requires the path to pass through an object a second time, and the object would be in a previously visited state were it not for the action of an event that induces a state transition. In this case, the "Examine the initial object model state" step can be neglected because its results have already been incorporated in the event tree model.
      v. A path is complete when
         (1) All objects in the system have completed responding to the system stimulus (initiating event), or
         (2) The model reaches a logical loop (here we must warn the analyst that a loop condition has been reached)
      vi. As each path is completed and its results recorded, the algorithm then recursively returns to complete the analysis of unfinished paths (generated by "splits") until all paths are complete.
   b. Method 2: All events and DAVs; this method is the same as Method 1, except:
      i. In each instance where in method 1 the technique looked for random events and split paths on that basis, method 2 *also* looks for any nonrandom events *and* DAVs that are external to the object and can affect the object (through the DFD). The

---

* Here "split" means the event in question becomes a top event ("question") in the event tree. The system trajectory path to that point becomes two separate paths, each of which must be resolved to completion in order for the event tree model to be complete.

** This method does not presently consider logical loops. Rules for dealing with such loops will be developed at a later time

method splits paths based upon each of these, in the order in which they are encountered in the model.

ii. Note that attributes are not required to be binary, but a single attribute may make use of several DAVs. Thus when the path is split based upon DAVs that can affect the object through the DFD, the path is split *n* ways at that point, where *n* is the number of DAVs associated with that attribute. Previously, for events, the path was split only two ways.

iii. Examine the path to see whether it remains within the restrictions specified in the list of valid states and conditions. If the path is found to be invalid, truncate it. This step is especially important in method 2 because invalid conditions are often defined in terms of physically incompatible combinations of DAVs.

iv. As each new object and/or state is entered, the same steps (from method 1, as modified above) are applied to that object or state. Paths are split as necessary.

v. All rules regarding logical loops, stop criteria, and the completion of unfinished paths are unchanged from method 1 when applied in this method.

c. Method 3: Split only based on specified events and/or attributes

i. This method is identical to method 2 except that the analyst specifies that the analysis is to perform path splitting only for particular events and/or attributes. All other events and attributes are held at their initial values unless those values are required to change by the operation of the model during the simulation of a path.

ii. All rules regarding path validity, logical loops, stop criteria, and the completion of unfinished paths are unchanged from method 2 when applied in this method.

A.3 FMEA and FMECA Development Rules

1. Specify an initial state for the system model.
   a. State of each object in the model
   b. All discrete attribute values

2. Define any invalid states and/or conditions.
   a. Valid versus invalid conditions
      i. List any states that are impossible or out of bounds.
      ii. List any combinations of DAVs that are impossible or out of bounds.
   b. Consider both invalid initial conditions and physically unrealistic operational conditions.

3. Form a list of scenarios (events and/or DAV changes) that are to be assessed in this analysis. Each should consist of a *single* event or DAV change that will be assessed by the tool one at a time.
   a. Include all model events that are classified as "component failure" — especially "random" events.
   b. Include all external events and/or attribute influences that could pose a threat to the system.
   c. Other events and/or DAV changes can be included at the analyst's discretion ("postulated scenarios").
   d. Ensure that each scenario does not initially place the system model into one of the invalid states and/or conditions defined above. If it does, drop the scenario.

4. For each scenario in the list constructed above:
   a. Set up the system model in the specified initial condition.
   b. Apply the condition(s) specified in the scenario.
   c. "Run" the object model based upon these conditions and stimuli until it reaches equilibrium.

d. Compare the trajectory and results produced by this run of the object model with the physically unrealistic operational conditions defined above. If it is physically unrealistic, drop the scenario.

e. If possible, compute consequences for the scenario.

f. Place the scenario definition, results, and consequences in a table for display.

Distribution

External:

Kevin Corker, Ph.D.
San Jose St. University Foundation
NASA Ames Research Center
MS 262-1
Moffett Field, CA 94035-1000

Mike Skroch
DARPA/ISO
3701 N. Fairfax Dr., 7th Floor
Arlington, VA 22203-1714

Tom Markham
Secure Computing Corporation
2675 Long Lake Road
Roseville, MN 55113-2536

Corky Parks
National Security Agency
Organization C12
9800 Savage Road
Ft. Mead, MD 20755-6000

Bill Unkenholz
National Security Agency
Organization C12
9800 Savage Road
Ft. Mead, MD 20755-6000

Dennis Longley
Queensland University of Technology –
Gardens Point Campus
GPO Box 2434
Brisbane, QLD
Australia 4001

Gary Luckenbaugh
Lockheed Martin Federal Systems Inc.
700 N. Frederick Ave.
Gaithersburgh, MD 20879

Bill Godwin
Lockheed Martin Mission Systems
1790 Los Pueblos
Los Alamos, NM 87544

Internal:

MS0188	D. L. Chavez, (LDRD Office), 4001
MS0405	M. P. Bohn, 12304
MS0405	R. J. Breeding, 12334
MS0405	T. D. Brown, 12333
MS0405	D. D. Carlson, 12304
MS0405	T. R. Jones, 12333
MS0405	K. J. Maloney, 12333
MS0405	C. G. Shirley, 12304
MS0428	W. C. Nickell, 12300
MS0449	P. L. Campbell, 6237
MS0449	J. H. Moore, 6234
MS0449	R. A. Sarfaty, 6237
MS0449	B. J. Wood, 6234
MS0451	J. Espinosa, 6238
MS0451	J. E. Nelson, 6235
MS0451	J. J. Torres, 6235
MS0451	R. E. Trellue, 6238
MS0455	L. R. Gilliom, 6232
MS0481	K. Ortiz, 2168
MS0482	M. J. Martinez, 2161
MS0490	W. H. McCulloch, 12331
MS0490	J. A. Cooper, 12331
MS0736	T. E. Blejwas, 6400
MS0741	S. G. Varnado, 6200
MS0742	J. R. Guth, 6414
MS0746	R. M. Cranwell, 6411
MS0746	D. J. Anderson, 6411
MS0746	M. E. Armendariz, 6411
MS0746	C. B. Atcitty, 6411
MS0746	J. R. Bechdel, 6411
MS0746	J. E. Campbell, 6411
MS0746	L. D. Chapman, 6411
MS0746	R. D. Lathon, 6411
MS0746	J. H. Linebarger, 6411
MS0746	B. E.Meloche, 6411
MS0746	D. P. Miller, 6411
MS0746	L. P. Swiler, 6411
MS0746	T. T. Sype, 6411
MS0746	D. G. Robinson, 6411
MS0746	B. M. Thompson, 6411
MS0746	H. O. Whitehurst, 6411
MS0747	A. L. Camp, 6412
MS0747	R. G. Cox, 6412

MS0747 V. J. Dandini, 6412
MS0747 F. A. Duran, 6412
MS0747 J. A. Forester, 6412
MS0747 J. L. LaChance, 6412
MS0747 L.A.Miller, 6412
MS0747 T. A. Wheeler, 6412
MS0747 D. W. Whitehead, 6412
MS0747 G. D. Wyss, 6412 [25]
MS0747 S. L. Daniel, 6412
MS0748 J. J. Gregory, 6413
MS0759 M. K. Snell, 5845
MS0759 C. D. Jaeger, 5845
MS0769 D. S. Miyoshi, 5800
MS0830 J. M. Sjulin, 12335
MS1138 D. R. Funkhouser, 6532 [2]
MS1138 B. N. Malm, 6532
MS1138 R. L. Vandewart, 6532 [2]
MS1345 P. A. Davis, 6114
MS1363 R. L. Craft, 6234 [20]
MS9011 F. B. Cohen, 8910
MS9018 Central Technical Files, 8940-2
MS0899 Technical Library, 4619 [2]
MS0619 Review & Approval Desk, 00111
For DOE/OSTI [1]